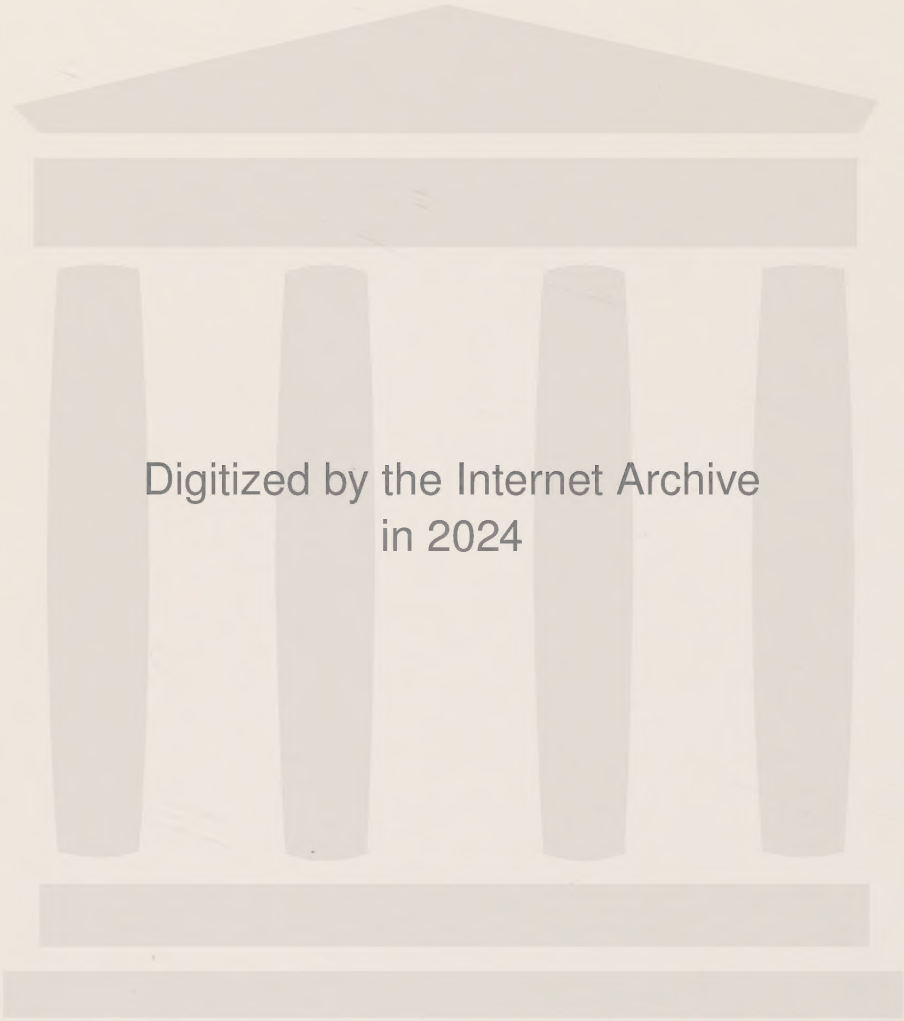
UNIVERSITY OF ARIZONA
39001028114778


Digitized by the Internet Archive
in 2024

https://archive.org/details/creatingapplicat0000bake

# Creating Applications with the IBM® OS/2® Extended Edition Database Manager

QA
76.76
.063
B345
1989

# Creating Applications with the IBM® OS/2® Extended Edition Database Manager

David C. Baker
William L. Banning
William W. Myre

AUG 22 1990

**Addison-Wesley Publishing Company, Inc.**

Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England
Amsterdam • Bonn • Sydney • Singapore
New York • Tokyo • Madrid • San Juan

The authors have made every effort to supply information that was complete and accurate as of the time this book was published. However, they assume no responsibility for its use, nor any infringements of patents or other rights of third parties which might result.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

C/2, IBM, Operating System/2, OS/2, Personal System/2, PS/2, and Systems Application Architecture are registered trademarks of International Business Machines Corporation.

DataEase is a registered trademark of DataEase International, Inc.

dBASE II, dBASE III, and dBASE IV are registered trademarks of Ashton-Tate Corporation.

Lotus 1-2-3, Symphony, WKS, WRK, WK1, and WR1 are registered trademarks of Lotus Development Corporation.

Paradox SQL Access is a registered trademark of Borland International.

**Library of Congress Cataloging-in-Publication Data**

Baker, David C.
Creating applications with the IBM OS/2 extended edition database manager.

1. OS/2 (Computer operating system) 2. Data base management. I. Banning, William L., 1945–
II. Myre, William W. III. Title.
QA76.76.063B345 1989 005.4'469 89-18110
ISBN 0-201-15302-5

Copyright © 1989 by David C. Baker and William L. Banning

This edition applies to Version 1.1 of the IBM OS/2 Extended Edition.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Carole McClendon
Developmental Editor: Joanne Clapp
Technical Reviewer: Philip Chang
Cover Design: Doliber Skeffington
Text Design: Kenneth J. Wilson (Wilson Graphics & Design)
Set in 10-point Caledonia by Compset, Inc., Beverly, Massachusetts

ABCDEFGHIJ-HA-89
*First printing*, October 1989

*To Bernie, who listened and encouraged.*

DCB

*Thanks to Lois, Tom, and John for putting up with the many things that did not get done during the preparation of this book.*

WLB

# CONTENTS


Foreword xiii
Preface xv
Acknowledgments xvii

PART ONE
Introduction to Database Manager Applications 1

Chapter 1. What is the Database Manager? 3
- Overview 3
- Component Relationships 4
- Database Services 5
- High Level Language Programs 6
- The Query Manager 7
- Conclusion 7

Chapter 2. What is a Database Application? 8
- Overview 8
- Elements of an Application 8
- What's Different about a Database Application? 8
- Elements of a Database Manager Application 10
- The Programming Language 10
- The Database Services Function Calls 11
- The Structured Query Language 13
- The Query Manager 14
- Conclusion 15

PART TWO
Database Application Design Considerations 17

Chapter 3. Database Structure 19
- Overview 19
- Database Services Objects 19
- Data Objects 20
- User Tables 20
- Indexes 24
- Views 27
- Access Plans 28

Naming Database Services Objects 29
Supporting Objects 30
Catalog Tables 30
Recovery Log 31
Database Configuration File 31
Database Security 32
Conclusion 32

**Chapter 4. Database Consistency** 33

Overview 33
What is Consistency? 33
Transaction Overview 34
Locking Overview 35
Logging Overview 35
How Locking Works 36
Types of Locks 36
Lock Transitions for a Single Transaction 37
Lock Compatibility among Several Transactions 40
Effects of Locking on Concurrency 41
Deadlocks 42
How Logging Works 44
Log Records 44
Log and Database Synchronization 44
Managing Log Space 45
Recovery 46
Conclusion 47

**Chapter 5. Database Configuration Parameters** 48

Overview 48
Database Services Configuration Parameters 48
Setting sqlnseg 50
Database Configuration Parameters 51
Lock Manager Parameters 52
Buffer Pool Manager Parameters 52
Log Manager Parameters 54
Storage Manager Parameters 55
How to Change the Database Configuration Parameters 57
Conclusion 57

**Chapter 6. Database Manager and OS/2** 58

Overview 58
Database File Layout 58
How Many Databases? 59
Volume and System Directories 60
Conclusion 62

PART THREE
Writing Database Programs in C 63

Chapter 7. Database Program Fundamentals 65

Overview 65
Database Program Model 65
Connect to a Database 66
Using SQL Statements 67
Stop Using a Database 71
Error Handling 71
The SQL Communications Area 72
Using the SQLCA 73
Error Messages 76
Handling Ctrl + Break 77
Supplied Include Files 78
Building a Sample Database Program 79
How to Precompile, Compile, and Link the Example 82
Conclusion 83

Chapter 8. Using SQL Statements 84

Overview 84
Defining Data Objects 84
Creating Tables 85
Creating Indexes 85
Initializing Table Data 85
Retrieving Data 87
Retrieving a Single Row 87
Retrieving Multiple Rows 87
Updating Data 92
Deleting Data 93
Conclusion 94

Chapter 9. Using Dynamic SQL 95

Overview 95
Understanding Dynamic SQL 95
Dynamic Statements other than SELECT 97
Inserting Parameter Markers 97
Using Dynamic SELECTs 98
Fixed List Selects 98
Varying List Selects 99
Data Types 105
Building a Sample Program 108
Hints 109
Conclusion 110

**Chapter 10. Designing Your Database Program** 111

Overview 111
Using the Database Application Process Model 111
Working with One Database 112
Data Handling 113
Managing Transactions 113
Determining the Length of a Transaction 114
Deciding When to Acquire Locks 114
Handling Waits 115
Handling Deadlocks 115
Handling Log Full Conditions 116
Designing the User Interface 116
Binding's Effect on Program Design 117
Conclusion 119

**Chapter 11. Precompiling, Compiling, and Linking** 120

Overview 120
Programs You Will Need 120
The Precompilation Process 122
Precompile and Bind 122
Precompile with Bind 124
Precompile without Bind 126
Compiling 128
Linking 129
Debugging 129
Conclusion 130

**Chapter 12. Utility and Environment Functions** 131

Overview 131
Coding the Utility and Environment Routines 131
Improving Performance 132
Reorganizing Your Database 133
Running Statistics 134
Exchanging Data with Import and Export 134
Specifying the Data Format 136
Specifying Columns 136
Environment Functions 138
Creating Databases 139
Erasing Databases 140
Displaying the Database Directory 141
Changing the Database Directory 145
Configuration 147
Conclusion 148

**PART FOUR**
**Supplementing Your Application with the Query Manager** 149

**Chapter 13. Query Manager Usage and Terminology** 151

Overview 151
Query Manager Objects 151
Using the Query Manager 152
The OS/2 Desktop 152
Starting the Query Manager 153
Screen Format 154
Screen Navigation 157
Query Manager Terminology 160
Conclusion 161

**Chapter 14. Query Manager Decision Support Tools** 162

Overview 162
The Ad Hoc Query Process 163
Starting the Query Process 164
Prompted Query 165
Using Variables 166
Saving Specifications 169
Form Specification 170
Multiple-column Usages 170
Breaks and ORDER BY 170
Multiple Column Heading Lines 172
Floating Point Data 172
Packaging Your Queries 172
Procedures 173
Using OS/2 Commands 175
Building and Exporting Queries 177
Conclusion 178

**Chapter 15. Customizing Query Manager** 179

Overview 179
When to Use Panels 180
Application Prototyping 180
Simple Applications 181
Personal Usage Applications 181
When Not to Use Panels 181
Applications for Others 182
Data Concurrency 182

Exploiting the Computer 182
Complex Applications 182
Complex Data Editing 183
Performance 184
Integrating Other Programs 184
Database Design 184
Editing LONG VARCHAR Fields 185
Using Panels 185
Structure 185
Variables 186
The Controlling Panel 187
Conclusion 188

**Chapter 16. Database Operational Support Tools** 189

Overview 189
Database Utilities 189
Database Backup 190
Database Restore 191
Table Utilities 193
Export 194
Import 198
Importing Data from DB2 or SQL/DS 203
Conclusion 203

**Appendix A. Database Services C Function Names** 205

Environment Routines 205
Utility Routines 206
Configuration Routines 206
Application Support Routines 206

**Appendix B. Sample Programs** 207

CATALGDB.SQC: Create a New System Directory 208
CREATEDB.SQC: Create a New Database 211
DBIMPORT.SQC: Import File into Requested Database 214
DBLIST.SQC: List System or Volume Directory 220
DBSTRUCT.SQC: List Tables and Views in Requested Database 223
ERASEDB.SQC: Erase the Named Database 228
FILLTB.SQC: Add Data Rows to Table Created by INSTALL 230
INSTALL.SQC: Simulate Application Installation Process 232
LOGFILE.SQC: Adjust Logfile Configuration Parameters 236
QUERY.SQC: Dynamic SQL Programming Example 240
RENAMEDB.SQC: Change Local Name for a Database 250
UNCATDB.SQC: Remove Entries from the Database Directory 253

**Index** 255

# Foreword

The OS/2 Extended Edition is one of IBM's most important undertakings, and I had the privilege of participating in several stages of its development. As a member of the Corporate Programming Staff, I helped define IBM's Systems Application Architecture (SAA). Then, as the Product Manager for the Database Manger, my team was responsible for the planning, design, and development of that component of the Extended Edition. Finally, as Systems Manager, my organization was responsible for all the components of OS/2 Extended Edition Version 1.1 (Database, Communications, and LAN). This set of experiences made it evident to me that the OS/2 Extended Edition Database Manager is a critically important offering in the following ways:

- For customers: It provides the same basic technology and capabilities found in IBM's leadership S/370 database products (DB2 and SQL/DS) at the individual and LAN server levels.
- For IBM: It completes a family of relational products across the breadth of IBM's product line.
- For the industry: It takes an additional step in implementing SQL.

This book is a must for those developing applications that access IBM's OS/2 Extended Edition Database Manager. It goes far beyond the reference books and user guides in the following ways:

- Gives clear explanation of the fundamental issues and trade-offs associated with relational database management systems.
- Explains the rationale behind the design choices and technology in IBM's relational database products and, in particular, the OS/2 Extended Edition Database Manager.
- Includes hints that offer powerful insights into developing the most efficient, best performing applications based on OS/2 Extended Edition Database Manager.

As a result, you can save valuable time gaining the expertise to exploit IBM's database technology on OS/2.

IBM pioneered SQL and relational database technology in 1970. Since then, IBM has continued to make major investments in relational database research. The result is a family

of database products that offers outstanding performance across the entire range of applications, from complex query optimization for decision support to quick updates for transaction oriented environments. For example, the OS/2 Extended Edition Database Manager uses algorithms and technologies found in DB2, but rewritten in C and optimized to the OS/2 environment. In some cases, Database Manager uses even more sophisticated techniques than DB2 made possible by advances in IBM research after DB2's original release (index locking, for example). The members of the IBM Relational Database Family are DB2 (for MVS), SQL/DS (for VM), Database Manager/400 (for OS/400), and OS/2 Extended Edition Database Manager. The Database Manager plays a major role both within OS/2 Extended Edition and within the framework of IBM's Systems Application Architecture. The OS/2 Extended Edition is intended to provide a distributed application platform that has been system tested by IBM; provides consistent user interfaces, publications, administration, configuration and installation; provides a single, reliable contact for education and support; and provides you with the function and performance you require across the full breadth of IBM distributed systems.

I wish you good reading and good luck creating outstanding OS/2 solutions for your customers!

*Pat Motola*

Manager of Business Development
Metaphor Computer Systems

# Preface

This book is for the individual who wants to design and build database applications for the IBM OS/2 Database Manager. It provides information and recommendations on how the technology in the Database Manager affects your application design. Additionally, examples are given of C language programs that use the Structured Query Language (SQL) and other functions of the Database Manager. The Query Manager is described in terms of how to best use its function to supplement your efforts to create a complete database application.

This book is intended for the reader who is already familiar with the Database Manager, has possibly installed and tried the Query Manager user interface, and is now wondering how to combine all of the Database Manager features into a database application. If you plan on writing database programs, we have assumed that you are familiar with the C programming language. If you plan to use the Query Manager to supplement your database programs, we have assumed that you are familiar with its general operation.

The discussion on Database Manager is divided into four parts. Part 1 provides an overall description of the Database Manager and an overview of the elements used to build Database Manager applications. You should read this part as well as Part 2, "Database Application Design Considerations," before starting to develop your application.

Part 2 discusses how the Database Manager implements the relational model. This part provides hints on designing your database application to take advantage of the Database Manager implementation.

Part 3 guides you through the use of SQL statements and Database Manager function calls in the C language. This combination of SQL and C function calls provides a powerful and flexible medium for developing sophisticated database applications. Programming examples are taken from a set of database programs that are listed in the Appendix.

Besides providing some basic guidelines on use of the Query Manager, Part 4 shows how Query Manager features can be exploited to supplement your C language database programs.

Appendix A lists Database Manager function names according to their functional category. This listing can tell you in a glance what functions you need to perform particular Database Manager functions such as create or connect to a database. Appendix B contains listings of the sample programs described in Part 3. You can use these C language programs as a database application toolkit. They provide useful functions such as listing the

tables, views, and columns defined in a database or importing a file to a table. Other programs contain helpful examples of SQL that you can use directly in your own database applications, such as a program that executes SQL statements entered onscreen.

## RELATED PUBLICATIONS

We have not tried to replace the IBM publications. Information in this book, such as the detailed database design considerations, sample programs, and Query Manager application design guidelines, supplements the reference material found in the IBM publications. Where possible, you will be referred to the appropriate IBM manual for reference material.

For writing database programs in C you will need *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference*, which is available separately from IBM as part number 90X7905. This guide contains the command and syntax reference for the Database Services functions. You will also need the IBM C/2 compiler and supporting documentation. Depending on what other features of OS/2 you plan on using, you may require additional software and documentation. For example, if you plan on using the Presentation Manager you will require The IBM OS/2 Toolkit.

For working with the Query Manager you will need *IBM OS/2 Extended Edition User's Guide*, which comes with the OS/2 Extended Edition. This guide contains step-by-step instructions on how to use the Query Manager.

Readers with complicated data management needs should familiarize themselves with the relational model and SQL. The following books are good sources of information on the relational model and SQL: *IBM OS/2 Extended Edition SQL Concepts*, *IBM OS/2 Extended Edition SQL Reference*, IBM part number 90X7907, and *A Guide to the SQL Standard* by C.J. Date (Addison-Wesley, 1987).

If you are struggling with the mass of IBM publications, you might order the *IBM OS/2 Information and Planning Guide*, form number G360-2650. This publication contains product and planning information about both the IBM OS/2 Standard and Extended Editions. Use it as a guide to the other IBM publications on OS/2.

# Acknowledgments

This book would not have been possible without the support and expertise of all the people at Addison-Wesley, in particular, Carole McClendon, Joanne Clapp, and Rachel Guichard. We also owe our thanks to Philip Chang for his thorough and thoughtful technical evaluation of the manuscript.

Clearly, the information in this book represents the conglomerate knowledge of many people, especially those IBM people who wrote the OS/2 Database Manager. These are the individuals who created this software and share in the creation of this book. Several of them offered suggestions and provided a perspective that helped form the contents.

We would also like to express our appreciation to the International Business Machines Corporation (IBM) for their assistance. All views expressed in this book are solely the authors' and should not be attributed to IBM or any other IBM employee.

# PART ONE

# Introduction to Database Manager Applications

Before beginning to develop your own Database Manager applications, you should be familiar with the functions that Database Manager offers. Part 1 of the book explains those functions.

Chapter 1, "What is the Database Manager?", provides an overview of Database Services and the Query Manager, the two major components of the Database Manager. This overview helps you understand the purpose of these two components and how you can use them to meet your data management needs.

Chapter 2, "What is a Database Application?", defines a database application in terms of the functions available to the Database Manager application developer. Each of the components described in Chapter 1 provides a specialized set of functions to be used in building your application. After reading this chapter, you should have sufficient understanding of the elements of a Database Manager application so that you can begin planning which elements to include in your database application.

# CHAPTER ONE

# What is the Database Manager?

## OVERVIEW

"What is the Database Manager?" is a simple question that requires a lengthy answer. This chapter begins to answer this question by describing the components available in the Database Manager, their relationship to the IBM OS/2 operating system, and how you access these components.

The IBM Database Manager is a full function relational database management system. It includes such features as concurrent data access, data consistency, data protection, and security. As shown in Figure 1-1, your data is presented in simple tables with information structured into rows and columns. The IBM Structured Query Language (SQL) is used to define, enter, and retrieve your data.

STAFF Table

| ID | NAME | DEPT | JOB | YEARS | SALARY | COMM |
|---|---|---|---|---|---|---|
| 10 | Sanders | 20 | Mgr | 7 | 18357.50 | - |
| 20 | Pernal | 20 | Sales | 8 | 18171.25 | 612.45 |
| 30 | Marenghi | 38 | Mgr | 5 | 17506.75 | - |
| 40 | O'Brien | 38 | Sales | 6 | 18006.00 | 846.55 |
| 50 | Hanes | 15 | Mgr | 10 | 20659.80 | - |
| 60 | Quigley | 38 | Sales | - | 16808.30 | 650.25 |
| 70 | Rothman | 15 | Sales | 7 | 16502.83 | 1152.00 |
| 80 | James | 20 | Clerk | - | 13504.60 | 128.20 |

**Figure 1-1** A Relational Table from the IBM SAMPLE Database

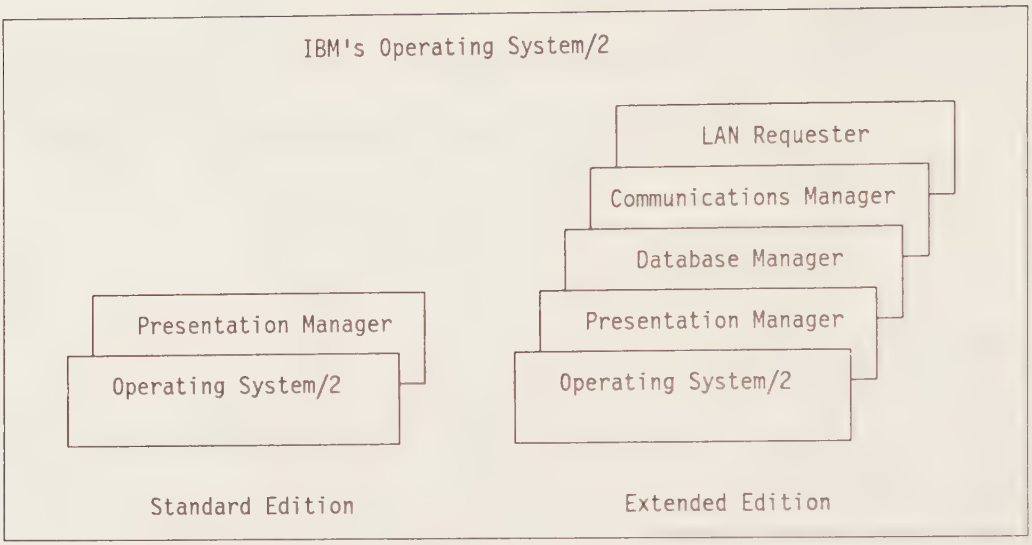



**Figure 1-2** IBM's Operating System/2 Offerings

## COMPONENT RELATIONSHIPS

The Database Manager is a subsystem of the IBM Operating System 2 Extended Edition. IBM offers two editions of its operating system, the Standard and Extended Editions, the Database Manager is available only in the Extended Edition. Figure 1-2 shows IBM's operating system offerings and the subsystems of the Extended Edition.

The IBM Database Manager provides two ways to work with your data. The first method is through programs written in high-level programming languages, such as C. These programs issue SQL statements and function calls that interface with the Database Manager. The second method uses the Query Manager, which provides an interactive interface to the Database Manager. You can develop applications using only one method or the other, but the most effective applications combine both programming and the Query Manager.

Figure 1-3 shows the relationship between the two methods of operation. As you can see, the Query Manager appears as an application program using the same services as any other database program. The interface that each application uses is called Database Services. Database Services processes both SQL statements and program function calls and is sometimes referred to as the database "engine."

The next three sections describe the features of Database Services and the two methods of accessing those features—high-level language programs and the Query Manager.

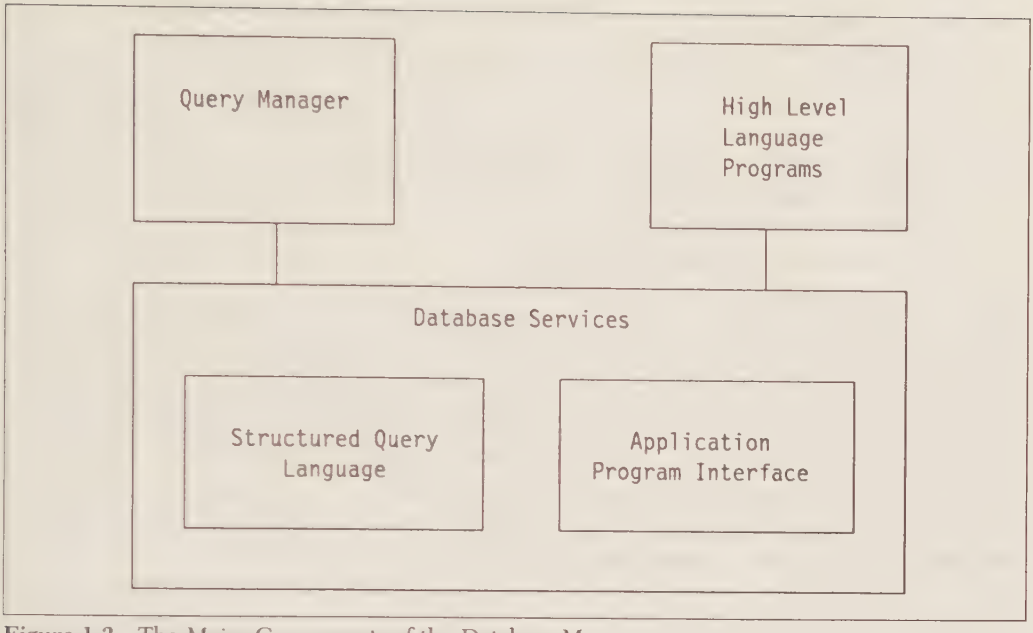



**Figure 1-3** The Major Components of the Database Manager

## DATABASE SERVICES

Database Services is the workhorse of the Database Manager. It stays behind the scene and provides the concurrency, data consistency, and protection inherent in the Database Manager. End users access Database Services through high-level language programs written to the Database Services interfaces or through the Query Manager. You should view Database Services as an extension of the OS/2 operating system function. A program calls Database Services to access database tables in much the same way that it calls OS/2 to access a file.

The features of Database Services include:

- **Structured Query Language**

  SQL statements can be embedded in application programs or entered into interfaces such as the Query Manager. These statements manipulate the data in your database.

- **Function calls**

  Functions that are unavailable in the Structured Query Language are provided through an application program interface (API) to Database Services. The function calls of this API allow access to database utility and configuration routines.

- **Precompiler/Binder**

  These utilities support programmers building database application programs that use SQL. The precompiler translates SQL statements within an application program into code understood by a programming language compiler. The binder ties an application program to a specific database by creating an access plan for that program. An access plan contains the specific steps that Database Services takes in order to access the data.

## HIGH LEVEL LANGUAGE PROGRAMS

The most flexible approach to building database applications is to use a high-level programming language such as C. Your own programs gain access to Database Services through the use of SQL statements embedded in the midst of actual program language statements. These SQL statements define and manipulate data in your database. Special elements of SQL allow programming languages to process the results of database queries by bringing the data values into your program. Other elements allow your program, while it is executing, to generate and process SQL statements.

There are several reasons why you might want to build an application program to access Database Services rather than use an existing user interface:

- **Exploiting the Computer**

  You may wish to write your own application if you have particular requirements that cannot be fulfilled by any existing user interface or application. Such requirements may include the need to access regular OS/2 files, Presentation Manager functions, OS/2 Extended Edition communications, or special data input/output hardware.

- **Complex Applications**

  If existing user interfaces have limited capabilities for producing complex applications, you may need the additional flexibility of a programming language in order to control the data concurrency and error recovery aspects of your application. Also, development of complex applications is facilitated by debugging facilities for programming languages that are more powerful than those provided for interactive user interfaces.

- **Alternate User Interfaces (Front Ends)**

  If the Query Manager user interface is insufficient for your needs, or you have been using a different database user interface, you may want to create an alternate interface to Database Services. Several vendors have already indicated that their database interface will be available on top of the Database Services platform. These include Paradox SQL Access, DataEase, and dBASE IV 1.1.

To develop database programs, you must understand the Structured Query Language and the Database Services programming interfaces and have a solid grasp of the procedure for building programs using these interfaces. Part 3, "Writing Database Programs in C," teaches you how to combine SQL and the Database Services programming interfaces to

access the Database Manager. Detailed information and examples are also provided on how to compile, link, and run these programs.

## THE QUERY MANAGER

The Query Manager provides an end-user interface to Data base Services and includes functions to find, report, and maintain information stored in your database.

From an end-user perspective, the Query Manager provides functions, or tools, to support three major user tasks.

1. **Database Maintenance and Configuration** You can use these tools to back up or restore databases, create or delete Database Services objects; import, export, or reorganize tables, and specify configuration parameters. Most users will find the Database Tools extremely helpful and supportive of almost any application developed to use the Database Manager.

2. **Decision Support** Many users find that the Query and Report tools provide a quick and easy way to access information in the database. You can use these tools in an ad hoc manner or with predefined queries to obtain information to help the decisionmaking process or for regular information reports.

3. **Customized Applications** The Procedure, Menu, and Panel tools provide a method to define simple applications for retrieval and maintenance of database information. These tools can provided a customized, application-specific, user interface to the functions of the Query Manager.

If you have simple data requirements that do not require application programming you may need only the Query Manager. The menu-driven interface of the Query Manager lets you enter data quickly and create meaningful reports with minimum database skills even if you don't understand SQL.

Part 4, "Supplementing Your Application with the Query Manager," discusses how you can use the Query Manager features to supplement your high-level language database applications to build a complete database application. Part 4 also provides guidelines on what types of applications might be built as Query Manager customized applications.

## CONCLUSION

Understanding the ways you can access your data is an important first step in building a Database Manager application. At this point, you may be wondering whether you should develop your application in a programming language or with the Query Manager. The next chapter provides an answer to this question by describing the elements of a Database Manager application. As you read the next chapter, you will discover that the question is not really a choice between the two methods but how to combine the best features of each interface.

# CHAPTER TWO

# What is a Database Application?

## OVERVIEW

This chapter explains what an application is and then shows how a Database Manager application differs. The four elements and associated capabilities of each Database Manager application are then described. After completing this chapter, you should be able to choose the elements you need to combine in order to build your own Database Manager application.

## ELEMENTS OF AN APPLICATION

Before we can define a Database Manager application, let's discuss the elements of a regular application. Regular applications can be broken into distinct functional aspects: input, decision control, calculation, data storage and retrieval, and output. These aspects are shown in Figure 2-1.

Database applications include these same elements, but differ in their methods of data storage/retrieval and decision control.

## WHAT'S DIFFERENT ABOUT A DATABASE APPLICATION?

In the past, personal computer applications used operating system-supplied methods for storing data. These functions were usually very simple, byte-oriented file input and output (I/O). Personal computer applications often used more sophisticated file management software for data storage and retrieval.

Database applications replace operating system file I/O with database I/O, which provides more than just data access. Database systems such as Database Manager include functions that were typically provided by the application writer. These functions include data sharing, transaction management, and recovery services.

Data sharing is facilitated since all database applications share the same structural view of data, the table format. Thus, two dissimilar applications can share data without needing to understand the logical structure of each program.

Data sharing allows multiple applications to access the same data concurrently while protecting each program from changes made by other programs. An application that uses OS/2 file I/O must provide concurrency by implementing its own locking schemes. Such an application must understand how much data to lock, how to perform retries, and how to detect deadlocks in order to provide the same level of concurrency as Database Services.

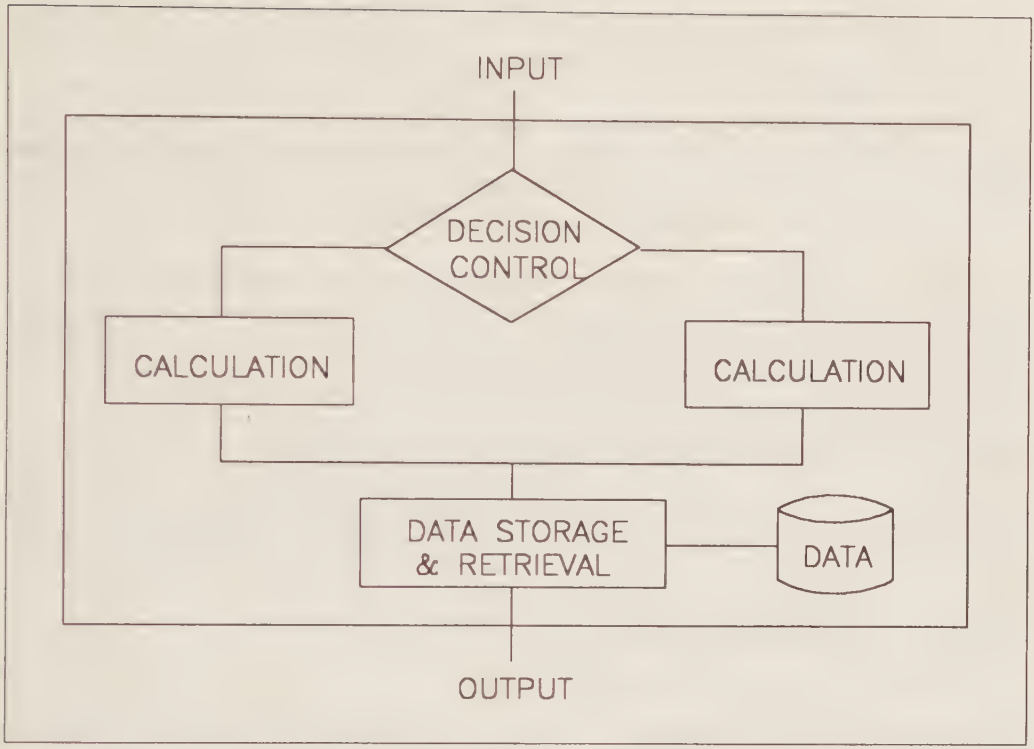



**Figure 2-1** Elements of an Application Program

With Database Services, merely accessing the data guarantees that no other program can change the data before your application completes. If another program is using a piece of data you require, Database Services waits until the data is available. Database Services also detects database deadlock situations for your application.

Database Services also provides commands to perform transaction management, which guarantees the consistency of your data. These commands let you determine how many and what type of actions make up a single, atomic operation. Database Services tracks the pending changes your application makes and, upon your command, either makes them visible to other programs (COMMIT) or discards them (ROLLBACK). "Transaction Overview" in Chapter 4 contains more information on this topic.

The main purpose of transaction management is to guarantee the consistency of your data through the use of a recovery service. If OS/2 terminates your program abnormally or if the power is turned off in the middle of a transaction, any unCOMMITed data will not appear in the database when it is restarted.

Relational database management systems such as Database Services also reduce the amount of decision logic you place in your program. The non-procedural nature of SQL combined with its set orientation means that you tell Database Services what you want

rather than how to get it. As a result, your application retrieves only the data it needs by restricting the amount of data returned on a SELECT statement (with the WHERE, GROUP, and HAVING clauses); you no longer program the data selection. Also, the use of VIEWS and JOINS in your application allows Database Services to combine data that you may have separated for data structure or administrative reasons. Another advantage of Database Services is that you can indicate the order in which you want the data returned. Thanks to the ORDER BY clause of the SELECT statement, you need not code program sorts.

A system such as Database Services simplifies the job of programming data management and frees you to concentrate on database applications rather than computer resource management. This book contains many techniques to help you understand and manage these new aspects of building database application programs.

## ELEMENTS OF A DATABASE MANAGER APPLICATION

Let's now turn our attention to the specific elements of a Database Manager application. In building Database Manager applications, keep in mind the four building blocks of the application:

- A high-level programming language
- The Database Services function calls
- The Structured Query Language
- The Query Manager

Each of these elements accomplishes specific tasks in the overall design of your database application. Certain tasks, such as report generation, can be accomplished using one of two different elements (you may either program the report or use the Query Manager). Later sections describe each of these elements so that you can decide which element best suits your application needs.

### The Programming Language

The programming language provides a framework within which you embed Database Services function calls and Structured Query Language statements. This framework allows you to control the sequence of your program statements, access operating system functions, and interact with the end user.

In essence, you use the programming language for everything except data storage and retrieval. A high-level programming language provides you the greatest flexibility in designing your database application because you can mix OS/2 system functions and Database Manager functions in the same application. This mix allows you to develop database applications that also use OS/2 file I/O to regular ASCII files, Presentation Manager functions for data display, and OS/2 Extended Edition communication functions for distribution of your application. Additionally, building an application in a high-level language lets you

exploit the computer hardware in ways that may not be possible from the user interface (such as special I/O devices like light pens and scanners). Query Manager applications do not provide this option.

The IBM Database Manager provides support for the IBM C/2 language in Version 1.1 of the OS/2 Extended Edition. Examples in this book are written in that language. You may try other vendors' C languages if you wish. IBM has announced its intention to support other languages, such as IBM COBOL/2 and Pascal/2.

## The Database Services Function Calls

The second element of your Database Manager application is the Database Services function calls. These function calls are embedded within your programming language and provide services other than data storage and retrieval, which are provided by SQL statements.

The Database Services function calls operate in a fashion similar to the standard C language function calls, like **printf**. Each function presents a call/return interface, and the calling program waits until the requested function completes.

The services provided by the function calls can be divided into four different areas:

- Environment Operations
- Utilities
- Configuration
- Application Support

These function calls are only available from within a high-level programming language. Following sections discuss the purpose of the function calls in each of the above areas. An indication is made where a particular function is also available from the Query Manager.

### *Environment Operations*

The Environment routines control databases. The five categories of routines are Database Connection, Application Environment, Database Directory Manipulation, Starting/Stopping Database Services, and Database Creation.

These routines deal with various aspects of the environment around your database but do not actually work with data within your database.

Of the categories listed above, the first two are most frequently used. The Database Connection routines are key in the development of database applications. Every application must use them in order to tell Database Services which database to use. The Application Environment routines are also critical to your application development. These routines are required to handle asynchronous interrupts from OS/2. Chapter 7, "Database Program Fundamentals," explains the use of the Database Connection and Application Environment function calls.

The manipulation of the database directories can be useful to your application if you wish to provide the end user with a selection of available databases. In most cases, however, your application either knows the name of the correct database or will ask the user

to supply the name before it begins its work. The Directory Manipulation function calls are described in Chapter 12, "Utility and Environment Functions," where sample programs are built that provide OS/2 command line interfaces to these functions.

The remaining Environment routines are used less frequently since the Query Manager provides an easy method for you to create and erase databases and maintain database passwords. A completely self-contained application may need to use these function calls in an installation program that creates the required database. Also, the Database Manager provides OS/2 commands for some Environment routines (Start and Stop Database Manager). You can use these routines from OS/2 command files (such as STARTUP.CMD) and avoid coding them in your application program.

### *Utilities*

The Utility routines are a group of procedures that cover a wide variety of functions from data exchange with other products to application program development support. The specific functions allow you to back up and restore databases, import and export data to from your database, reorganize database tables, and generate statistics on tables and indexes in your database. All of these utilities are also available from the Query Manager user interface.

You can perform backup and restore operations conveniently from the Query Manager rather than building these operations into every database application that you create. This is not a difficult task for users to perform, and they need not know anything of the internal structure of a database in order to perform a backup or restore. In some cases, you might want to program these functions in your application to provide an interface consistent with the rest of your application.

Similarly, data import and export are most easily performed from the Query Manager. Unless your application has a specific need to program a data import or export (as might be the case if you were developing an alternate front end to Database Services), you should consider leaving these tasks to the Query Manager. Even though you can access all of these utility functions from the Query Manager, there are reasons why some should be used only in a database application program.

Your program should provide a means of reorganizing application tables and regenerating database statistics on a timely basis in order to provide the best performance available to your application. By performing the reorganization and regeneration of statistics within your application program, the user need not know the names of all the tables and indexes that make up the database. This maintenance can be done without the end user knowing it is taking place. Alternatively, your application can provide the option for the user to request reorganization (without needing to specify the tables).

Sample programs that use these Utility routines are described in Chapter 12, "Utility and Environment Functions."

### *Configuration*

The Configuration routines allow your application to manipulate the Database Manager configuration files. There are two types of configuration files.

1. **Database Manager Configuration** Contains parameters that affect the performance of the system as a whole and the amount of certain OS/2 resources to be allocated when Database Services is started.

2. **Database Configuration** One for each database. Contains parameters that affect the performance when connected to a specific database and the amount of certain OS/2 resources to be allocated when an application connects to that specific database.

You should rarely need to use these function calls in a database application. If you wish to set any of the parameters, Query Manager should be sufficient for the task since changes are made infrequently.

Since you do not know what other applications may be running, do not attempt to set the Database Manager configuration parameters. Setting these parameters without regard to other applications (including non-database applications) can bring about undesirable side effects, such as diminished performance in a totally unrelated application.

The previous paragraph illustrates a concern of administering a Database Manager system. The lack of an intelligent configurator can cause problems if you want to change any of the configuration parameters. A good use of the Configuration routines might be to build a program that uses the configuration parameter relationships described in Chapter 5, "Database Configuration Parameters," to assist the end user in setting the system configuration.

### *Application Support*

The Application Support routines simplify the development of your database application; they have no counterpart in the Query Manager. The functions available to your program include translation of return codes into text messages for display, interruption of database requests, and generation of access plans, which are required during installation of your application. Use of the Application Support routines is covered in Chapter 7, "Database Program Fundamentals."

## The Structured Query Language

The third element of your Database Manager application is the Structured Query Language. SQL provides the statements that your program uses to create, store, and retrieve data.

SQL is not itself a programming language. The language is non-procedural by design, but it requires the procedural control of a host language to process results. You build your application by combining the decision and sequence control of the C language with the data storage and retrieval capabilities of SQL.

Since language compilers such as the IBM C/2 compiler cannot interpret SQL statements, IBM supplies a precompiler program that can translate your Structured Query Language statements into IBM C/2 language function calls. Chapter 11, "Precompiling, Compiling, and Linking," describes the IBM precompiler and how it builds executable database applications.

SQL can be used in two different manners within a program. One method is called *static* SQL because the statement is fixed (that is, actually typed into your program source) when you write your program. The other method is called *dynamic* SQL because the SQL statement is constructed at run time. For example, the Query Manager uses dynamic SQL since it knows neither the type of SQL statements you plan to execute nor the names of the databases or tables you plan to access.

The Structured Query Language is composed of Data Definition, Data Manipulation, Dynamic SQL, and Precompiler Support statements.

The Data Definition and Manipulation statements are the same whether you enter them into the Query Manager SQL Query Window or place them into an application program. Dynamic SQL and the Precompiler Support SQL statements are used only within a high-level language program.

This book is not a reference for the Structured Query Language. Information on how to use SQL statements in your database application is contained in Parts 2, 3, and 4.

## The Query Manager

The fourth element of your database application is the Query Manager. Development of certain aspects of your database application can be simplified through the use of the Query Manager. Previous sections identified several areas in which maintenance functions were easier to perform through the Query Manager. These areas include database creation, deletion, backup and restore, as well as data import and export. These are all infrequently performed functions, and the Query Manager operational support tools can supplement the more specific tasks of your database program.

Two areas in which the Query Manager can supplement your database program are report writing and ad hoc queries. The task of report writing can be tedious in a programming language. The Query Manager decision-support tools provide a menu-driven interface for definition and generation of reports. Query Manager procedures may be built to extract and print data according to a stored report format. Maintenance of these reports is much easier under the Query Manager than in a programming language—in fact, no programming skills are required in order to generate a Query Manager report.

Most often, your database application asks a specific set of questions of the database. These are the queries that you embed in your application program. If a user wants to ask a question that you have not included in your program, the Query Manager decision-support tools provide an interface for users to enter such ad hoc queries, as a result, you need not provide such an interface as part of your application. Even if they do not understand SQL, Query Manager users will be able to generate a query and receive the report.

The Query Manager customization functions can eliminate altogether the need for application programming. However, since these functions are of limited power, you should consider carefully the types of applications that can be built with these customization functions. See Chapter 15, "Customizing Query Manager," for guidelines on when to use the Query Manager customization functions.

## CONCLUSION

As you can see, a database application is really a combination of all the different Database Manager elements acting in concert. Such an application might include a high-level language program using embedded SQL and the Database Services function calls to provide a graphics interface to a database. The same application may have reports generated by the Query Manager decision support tools while requesting the user to back up or restore databases using the Query Manager operational support tools.

You should not rush off and start writing your programs or building your Query Manager queries and reports just yet, though. You must first understand how the Database Manager works and how that influences the design of your database application, topics that are covered in Part 2.

# PART TWO

# Database Application Design Considerations

Before developing your database application, you need a good understanding of the inner workings of the Database Manager. This part of the book is meant to give you that understanding.

No matter how you decide to develop applications for the Database Manager, you should read Chapter 3, "Database Structure." This chapter describes the objects you work with from both the Query Manager and a database program. This chapter contains hints on effective use of these objects.

Chapter 4, "Database Consistency," describes how the Database Manager maintains the consistency of your data. You should read the overview sections of this chapter for a good overall understanding of the consistency mechanisms. Remaining sections of this chapter provide detailed information that allow you to understand the environment in which your program executes.

Chapter 5, "Database Configuration Parameters," discusses how you can adjust the operation of Database Services. This chapter provides information on Database Services configuration parameters and how they can affect your application. Recommendations are made on appropriate settings of these parameters.

Chapter 6, "Database Manager and OS/2," explains the Database Manager's relationship to the underlying OS/2 operating system. Information in this chapter helps you put your application and Database Services into perspective.

# CHAPTER THREE

# Database Structure

## OVERVIEW

This chapter discusses the objects that make up a Database Services database. Some of these objects help you store and retrieve data. Tips are provided on choosing the most appropriate data types to hold your application data. Other objects play a supporting role. As an application developer, you should pay particular attention to these supporting objects because they deal mainly with aspects of data management that should be hidden from the end user. The chapter ends with a brief discussion of database security.

## DATABASE SERVICES OBJECTS

Database Services objects can be divided into two categories: *data objects* and *supporting objects*.

Data objects store and manipulate your data. The name and function of each data object follow.

| NAME | FUNCTION |
|---|---|
| User Tables | Contain your data |
| Indexes | Aid in effective access of table data through the use of keys |
| Views | A table that does not physically exist, but represents a portion (possibly all) of one or more underlying user tables |
| Access Plans | Contain information Database Services uses to access your data |

Supporting objects support access to user tables and views without actually containing user data. The name and function of each supporting object follow.

| NAME | FUNCTION |
|---|---|
| Catalog Tables | Record and control the database structure |
| Recovery Log | Records changes made in the database and supports transaction management and recovery from system failure |
| Database Configuration | Controls the amount of Database Manager resources used by the database |

All Database Services data objects and supporting objects are discussed in detail below.

## DATA OBJECTS

Data objects store and manipulate data. The data objects in a database are tables, views, and indexes. The table holds the data, and the views and indexes aid in controlling and accessing the table data.

### User Tables

The table is the fundamental data object of Database Services. All user data is stored in and retrieved from one or more tables in a database. The other data objects exist to enhance or support access to data stored as rows in tables.

A table has two elements: *table structure*, which is a definition of the columns in the table, and *table data*, which are the rows of data.

The table structure is specified when you use the CREATE command to create a table. You must create this table definition before inserting any data rows. These user tables, which actually contain data, are sometimes called *base* tables in order to distinguish them from other tables associated with Database Services, such as the *result* tables returned by SQL SELECT statements or from views, which are accessed like tables.

When creating a table, you specify each column with a unique name within the table and a data type. You will later use those column names to access the data in that table. Although the columns in a table are stored in one order, you may specify the column names on a SELECT in any order.

Once the table has been defined, rows of data may be inserted. Each row consists of a sequence of data fields corresponding to the columns of the table. The rows of data are unordered and are inserted wherever there is free space, although in practice the rows inserted into a new table are inserted in order. Database Services does not guarantee any order to the rows unless an ORDER BY is specified on the SELECT.

#### *Column Data Types*

You can define a column as one of the ten following data types:

- CHAR
- VARCHAR
- LONG VARCHAR
- SMALLINT
- INTEGER
- DECIMAL
- FLOAT
- DATE
- TIME
- TIMESTAMP

MENU

| SANDWICH | SMALL | LARGE | RANK |
|---|---|---|---|
| Roast Beef | 2.00 | 2.50 | 2 |
| Turkey | 2.00 | 2.50 | 3 |
| Ham | 2.00 | 2.50 | 7 |
| Mixed Cuts | 2.00 | 2.50 | 8 |
| BLT | 2.00 | 2.50 | 6 |
| Chicken Salad | 2.10 | 2.95 | 4 |
| Bologna | 1.65 | 2.10 | 10 |
| Cheese | 1.85 | 2.25 | 14 |
| Avocado | 2.10 | 2.95 | 13 |
| Tuna | 2.05 | 2.60 | 1 |
| Egg Salad | 1.75 | 2.25 | 12 |
| Salami | 1.65 | 2.10 | 9 |
| Seafood | 2.20 | 3.25 | 11 |
| Club | null | 3.25 | 5 |

**Figure 3-1** An Example Table of Sandwiches

One of the first steps in designing your application is to choose the appropriate data types for your table columns. This decision is complicated by the availability of multiple data types that, at first glance, seem to be similar in capability (for example, CHAR, VARCHAR, and LONG VARCHAR). Further complications are introduced by data type modifiers such as NOT NULL and FOR BIT DATA. The following section discusses how to choose among some of the more frequently used data types. A small example is presented, followed by some guidelines for choosing one data type over another.

The example table in Figure 3-1 contains four columns of information about sandwiches:

Sandwich—the name of each sandwich (type CHAR or VARCHAR)
Small—the price of a small sandwich (type DECIMAL)

Large—the price of a large sandwich (type DECIMAL)

Rank—the rank of each sandwich in sales (type INTEGER)

The Sandwich column contains text and can be either a CHAR or VARCHAR type. The difference between these two types is that CHAR is fixed length (you must specify the length of a CHAR column when creating the table) but VARCHAR supports variable length text fields (up to 4000 characters). Although you might assume that VARCHAR is a good choice since it seems more flexible, there are some trade-offs that are not immediately apparent. Let's investigate the advantages and disadvantages of CHAR and VARCHAR.

One disadvantage of CHAR is that it takes up a fixed number of bytes of storage whether or not anything useful is stored in a particular field. For example, assume that the Sandwich column in the example table is defined as a fixed length CHAR of 14 characters. Since the longest name is 13 characters long (Chicken Salad), all the names fit and are left justified with trailing blanks. However, the trailing blanks are wasted space. In this table, the total number of characters in the fourteen sandwich names (more precisely, the 14 rows of data) is 95 characters. However, the table requires 196 bytes (14 rows * CHAR(14)) to contain those fourteen sandwich names. Therefore, 101 bytes are wasted as trailing blanks. This wasted space is over half the total space used by the column.

If this table were 100K rows long and the space used in the column stayed at the same ratio (95/196 = 48% meaningful characters), then over 10 MB (52% * 196 bytes * 100K rows) of space would be wasted in this column alone (the wasted space is also called *overhead*). As you can see, the CHAR data type wastes space if the text varies in length and/or the average length is much smaller than the maximum.

VARCHAR also has overhead, but in a different way. VARCHAR columns use only the number of bytes actually passed by the application program but require an extra 4 bytes of overhead on each field to manage the variable length. Thus, in the example table, making Sandwich a VARCHAR requires the same 96 bytes for the actual data plus an overhead of 4 bytes for each of 14 rows for a total of (96 + 48) 144 bytes. When this table grows to 100K rows, the overhead is 400K (4 extra bytes * 100K rows), which is smaller by a factor of twenty than when the CHAR definition was used. Clearly, VARCHAR is the best choice for the Sandwich column, especially as the table grows larger (although it is hard to imagine a table of 100K different sandwiches!).

The following guidelines will help you choose between CHAR and VARCHAR.

- Use VARCHAR if the difference between the average length and the longest length is 4 bytes or more.
- Use VARCHAR if the defined maximum length of a column may increase, since increasing the size of any column requires that the table data be exported, the table dropped and recreated, and then the data imported back into the restructured table.
- Use CHAR if the text is all the same length and will not change.
- Use VARCHAR if suffix searching is desired. CHAR columns are padded with blanks, making suffix matching difficult.

| Data Type | Cost | Limits |
|---|---|---|
| CHAR(n) | n characters + N | 1 < = < = 254 |
| VARCHAR(n) | actual size + 4 + N | 1 < = n < 4000 where n is maximum size |
| LONG VARCHAR | 24 byte pointer + N | up to 32700 characters (stored outside the table) |
| INTEGER | 4 + N | −2,147,483,648 to +2,147,483,647 |
| SMALLINT | 2 + N | −32,768 to +32,767 |
| DECIMAL(m,n) | (m + 2)/2 + N | 1 < = m < = 31, m digits |
| FLOAT | 8 + N | IEEE 8 byte floating point |
| DATE | 4 + N | |
| TIME | 3 + N | |
| TIMESTAMP | 10 + N | |

Note:
N = 1 if data type declared nullable
N = 0 if data type not nullable

**Table 3-1** Column Data Type Overheads and Limits

- If a VARCHAR column is used within an index, the defined maximum for that VARCHAR is used for key length calculations. This expansion can limit the available column combinations you can have with a VARCHAR since total key length is limited to 255. This limit is derived from internal column lengths. For example, a nullable VARCHAR key with a maximum length of 100 uses 105 bytes of the index limit. See Table 3-1 for the internal sizes of other data types.

Since the amount of disk space used by VARCHAR columns is relative to the actual number of characters stored, it may seem that you should define all VARCHAR columns at the maximum length of 4000. But each row has an upper limit of 4005 characters and the row limit checking assumes that all columns are at their maximum lengths. Thus, you can have only one VARCHAR column defined at the maximum size of 4000. Of course, you can have multiple, smaller VARCHAR columns as long as the total row size does not exceed 4005. Consider using LONG VARCHAR columns if you must have more than 4005 varying characters per row.

The NOT NULL (called *Data Required* by the Query Manager) attribute applies to all data types and means when a row is added to a table, columns with the NOT NULL attribute must have a data value. Columns without this attribute can be added with a special indicator set that indicates the column has no data value. This is referred to as a NULL value in SQL terminology. Note that NULL is not the same as a field containing blanks or zeros. This requirement is also checked whenever the column is changed. The last row in Figure 3-1 contains a NULL value indicating that there is no small Club sandwich.

Although character fields generally contain displayable character information, you can store non-text data in character fields. In this case, the data is in a user-defined format and should not be translated or converted when the data is moved to another computer system where the internal representation of character data is different. For example, suppose you have data that represents a high resolution image of a map. The data is encoded and compressed and is intended to be sent to a graphics processor for display rather than directly to a display as character information. For data of this type, you should specify the FOR BIT DATA option on the column's definition. This fact is reflected in the Catalog Tables, where applications may access it in order to determine whether any data transforms should be made. Database Services uses this indicator to know that it should not check code page consistency of FOR BIT DATA columns on Import operations.

You should consider the amount of accuracy required for numeric data. A FLOAT column does not provide as much accuracy as a DECIMAL column, thus, you should store monetary data in DECIMAL fields. If your application accepts or displays numeric data in an ASCII format, consider using DECIMAL instead of FLOAT data types. Accuracy can be lost in the conversion from ASCII numeric representation to floating point. You will also find that moving floating point data from one computer hardware architecture to another may result in a loss of accuracy. For example, data moved from an IBM S/370 hardware architecture to an IBM PC machine does not retain its accuracy.

### *Data Type Overheads and Limits*

A table row can contain a maximum of 4005 bytes of data. Each column you define consumes some of those 4005 bytes. The amount of storage required for each data type is called the *cost*. This cost includes both the storage required for the data value plus any required overhead. Each data type also has some associated limits on the range or size of the data placed in a column of that type. Table 3-1 lists these costs and limits for each SQL data type.

You should use these values to assist in your table design. Choose an appropriate data type for the kind of information you plan to store in each column.

## Indexes

The purpose of an index is to help you locate information quickly. You are probably familiar with using an index in a book. Since the index entries in a book are listed in alphabetical order, you can scan the list and quickly find the specific topic you are interested in. Associated with the index entry is a page number that indicates where to look for information on that topic.

Database Services' primary use of indexes is to help you locate rows in a table quickly. Indexes also allow greater concurrency when more than one transaction is accessing the same table. These benefits are not without a price, however, since indexes increase the requirements for disk space and decrease performance when writing data to a table. This section describes how Database Services uses indexes and provides information that will assist you in making effective decisions regarding index use.

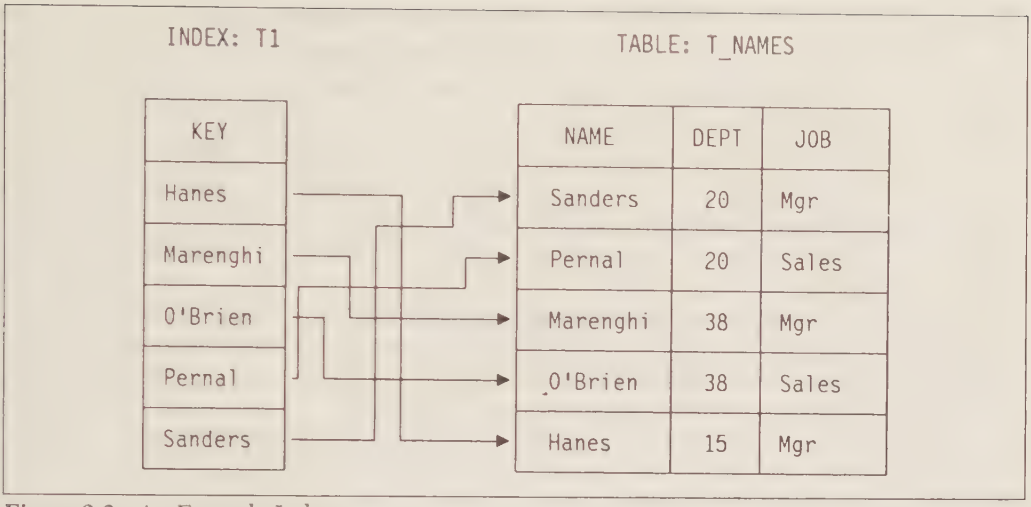



**Figure 3-2** An Example Index

### *Creating Indexes*

Although an index is defined on one and only one table in a database, each table can have multiple indexes. Since an index is dependent on the existence of a particular table, the table must already exist before the index is created.

An index consists of one or more columns from the table, specified in some order. These columns define the key. If there is more than one column in the index key, the columns are ordered from left to right in decreasing significance. Values for these keys are extracted from the table, sorted in some order, and stored in the index. Total key length is limited to 255 bytes (derived from internal column lengths).

Each key has an associated pointer to the row it indexes (see Figure 3-2). Unlike some PC data management systems that allow you to reference rows with a record number, the pointer is not accessible to your Database Manager application. Database Manager rows are selected based upon the values they contain rather than by record number.

When the specified key value is not unique within a table, an identical index key is created for each row containing the key value. If you are following formal relational database design principles, be sure that each of your tables contains a unique identifier (that is, each table should contain a key whose value in any row is unique within that table). This type of key is called a *primary key*.

Database Services can enforce uniqueness on key values through an index created as UNIQUE. When a UNIQUE index is created, the table is checked for rows with duplicate key values. If duplicate keys exist, the index is not created and an error is returned. Subsequently, any INSERT or UPDATE on the table that would create a duplicate key is not allowed, and an error is returned on those SQL operations. For completeness, all columns specified in the unique key should be created as non-nullable since the formal definition of primary key excludes NULL values.

In the example index in Figure 3-2, the Name column has been specified as a key for table T_NAMES. Note that the Name column values are duplicated in the index T1. Database Services stores these index entries in a special data structure called a *B+ tree*, which allows you to locate a particular value quickly.

Suppose we wish to find the department for employee "Hanes." Database Services accesses the index T1 and finds the first key value equal to "Hanes." Associated with this first matching index entry is an internal pointer into T_NAMES that points directly at the desired row. This row is returned and the index is scanned for other matching keys. Since there are no others, the scan completes.

Notice that the only time the table is actually referenced is when a matching index entry is found. This ability to reference the table only when a specific row is found provides the foundation for multiple applications to concurrently access the same table. The following section describes how indexes affect concurrent access.

### *Indexes and Concurrent Table Access*

One of the first things Database Services does when processing a request for data is to determine if an index exists for the table(s) being accessed. If no index exists, the entire table is locked. This table level lock decreases the ways other applications can access the table. In the case of a SELECT, a share lock is obtained on the table thereby allowing other applications to read the table but not write to it (INSERT, UPDATE, or DELETE). Other applications will be allowed to write to the table when the transaction owning the share lock ends.

The entire table is locked because, without an index, Database Services must read every row in the table in order to make sure your request is satisfied. This process of reading every row is called a *table scan*. If an index exists that matches the sequence of rows you want or the selection criteria (WHERE clause), Database Services can scan the index to retrieve the rows. This process is called an *index scan*. When row access occurs on an index scan, only the rows affected are locked. It is very important to note that concurrent access to that table is reduced or eliminated unless an appropriate index is defined on a table and is used on a scan. If your application has multiple (concurrent) users reading and writing the same tables, you must define an appropriate index.

### *Cost of Indexes*

The use of indexes on tables is not without cost, chief among which is increased disk space requirements. In some cases this cost may be greater than the benefit of performance increase. Indexes are physical objects separate from the table with which they are associated. Since the column(s) making up the index key exist in both the index and the table, indexes can rapidly increase the amount of disk space required for your database. For example, if a column were included in three keys, that column's data would be replicated in four places—three indexes and the table itself. It is important to remember that each index uses up disk space proportional to the size of the table on which it is defined. If you

created several indexes with multiple column keys on a table, the disk space used by the indexes could easily double or triple the space originally taken by the table.

Another consideration in using indexes is the time required to change a row. Three steps are involved in this process.

1. **INSERT**

   The inserted row is put in the table and its key value is put into each index.

2. **DELETE**

   The deleted row is removed from the table and the associated key is removed from each index.

3. **UPDATE**

   The updated fields of the row are put in the table. If the updated fields are part of a key, they are also updated in the appropriate index.

There is no penalty for locating rows; a penalty occurs only when you are actually changing data. Each index adds another round of changes when a row is inserted or a key value changed. Thus, you should limit the number of indexes to those you really need.

If you have an application that performs bulk updates or imports data and you want to run faster, consider issuing a DROP for each index before running your application. Removing the indexes avoids the additional updates required to keep the indexes current during your update activity. However, you should consider the extra time needed to recreate the index and run statistics when your application completes.

## Views

A view allows you to see part of, but usually not all of, something. In the case of Database Services, a view allows you to see some of the data in one or more tables. Views do not have to be defined on tables; they can be based on other views. A view is defined by a SELECT clause used with the CREATE VIEW statement. You can also create a view with the Query Manager.

You can read a view like a table (for example, in a SELECT statement). You can also update views (such as INSERT, UPDATE, or DELETE) if the view definition obeys some rules. To update a view, be sure that the view

- is defined on just one table or updateable view (no joins!)
- references only the column names from the table (no expressions can appear in the column list)
- does not use specifications like GROUP BY, HAVING, or DISTINCT
- does not contain a self-reference in a subquery clause

Details of these limitations on updateable views can be found in the *IBM OS/2 Extended Edition SQL Reference*.

You cannot define indexes on views because a view does not physically exist in the database like a table, and an index contains physical pointers in to the table for which it is defined. However, if appropriate indexes are defined on the table(s) making up the view definition, they will be used to access the table data through that view.

Now that we have defined views, you should know when and how to use them.

1. Views can be used to hide data that is in the table(s) on which the view is defined. Be aware, however, that table data hidden by the use of a view is not secure. If you have access to the view, you also have access to the underlying table(s).

2. If a view can be updated, it is possible that the update will not appear in the view. This situation occurs when a row is changed (or inserted) and it does not meet the constraints specified on the view definition. You can prevent this situation by using the WITH CHECK OPTION on view definitions. The WITH CHECK OPTION checks updates against the view definition before allowing an INSERT or UPDATE. The INSERT or UPDATE is not allowed if it does not match the constraints on the view (the WHERE clause in the CREATE VIEW). Thus, an updateable view can provide range checking in SQL (range checking is not available in an INSERT or UPDATE of a table). The overhead of using the INSERT or UPDATE commands through a view is minimal.

3. You can use a view to retain an image of a table after the original table is restructured. As a result, existing applications can be converted to run with little or no effort. When converting an application in this manner, you must pay attention to the update restrictions on views. If your application reads and writes data, the view you create to represent the old structure must be updateable; otherwise, you must convert the application to write to the new table structure.

### Access Plans

Any program that accesses a database does so through an access plan. An access plan contains information Database Services needs to access your data. If an application uses the database, it must be programmed with some SQL statements embedded in the program. When the application is converted to an executable object, the strategy for each SQL statement is stored in an access plan. This strategy includes such information as what indexes to use and how to use them.

Access plans are stored in the SYSPLAN and SYSSECTION Catalog Tables. We discuss them separately here as they constitute a database object independent of their storage in the Catalog Tables. The original text of the SQL statement is stored in another Catalog Table, SYSSTMT. By saving the statement text, Database Services can recover it if an automatic rebind is necessary or if it is to be returned as part of an error message.

You should be concerned with access plans only if you are writing a database program. When developing database programs, you should hide the creation of access plans. For example, the Query Manager has an access plan that it creates when you first use it to

work with a database. The act of an application creating an access plan for itself is called *binding* to a database. Access plans and binding are discussed in more detail in Chapter 11, "Precompiling, Compiling, and Linking."

### Naming Database Services Objects

Since you are familiar with the Database Services objects, you should now consider their names. Objects are named when they are created; these names are then used in SQL statements to refer to the object.

Object names consists of two parts. The first (leftmost) part is called the *qualifier*, and the second (rightmost) part is referred to as the *simple* (unqualified) name. The qualifier contains from one to eight characters. The simple name can be from one to eighteen characters, depending upon the object type. Syntactically, these two parts can be concatenated as a single string of characters separated by a period (".") When concatenated in this fashion, the object name is said to be "qualified." For example:

```
CREATE TABLE PAYROLL.STAFF..
CREATE TABLE STAFF....
```

The first CREATE TABLE statement shows a qualified name, and the second one shows a simple name. At a minimum, a simple name must always be specified in a SQL statement.

The qualifier provides a way of avoiding naming conflicts when your objects are stored in a database with another application's objects. For example, two applications can each have a table named PHONE, but each must be identified by a unique qualifier, such as PAYROLL.PHONE and LEDGER.PHONE.

The following terms are used synonymously with qualifier.

- AUTHID
- Authorization ID
- Collection name
- ID for Lists

You need not specify a qualifier. If a qualifier is not specified, the value of the OS/2 environment variable SQLUSER is used. If SQLUSER is not defined, the value "NULLID" is used. SQLUSER and qualifiers should be used only to group objects together. They do not imply any form of ownership or access control.

In designing your application, you must choose names for your database objects. If the object names are made visible to your end user, choose names that make it easy for a person to remember what the object contains. The user may be exposed to these names if you decide to use Query Manager for part of your application.

Always use qualified names when creating your objects and in your SQL programming statements. This practice ensures that the object names will not change if your user sets the value of SQLUSER. If you use static SQL statements that reference the qualified

object name, you will not need to determine the SQLUSER value and build your SQL statements dynamically.

Choosing a value for the qualifier does not matter if you require the user to keep your application in its own data base. All objects in the database are under your control and you can ensure that no naming conflicts arise. On the other hand, one of the major reasons databases are attractive is that they allow sharing of data between applications and users.

If your application shares a database with others, you must avoid an object naming conflict with those other applications. The problem is how to choose an appropriate qualifier. Some applications use their company's name or some part of it for the qualifier. This convention can be adopted by your own and purchased applications. Restrictions on qualifier names are covered in the *IBM OS/2 Extended Edition User's Guide*.

## SUPPORTING OBJECTS

Supporting objects facilitate access to tables and views without actually containing user data. Supporting objects include the Catalog Tables, recovery log, and database configuration file.

### Catalog Tables

All data management systems, including the Database Manager, need to access and store housekeeping information such as table names. This information is stored in Catalog Tables in each database. Catalog Tables are somewhat different from ordinary user tables in a database. Although they can be read with a SQL SELECT just like any other table, they cannot be created, changed, or deleted by a database user. The names of the Catalog Tables are reserved and cannot be used for other purposes. Catalog Tables are referenced using the SYSIBM qualifier.

Each database has its own set of Catalog Tables that are created automatically when the database is created. They are also updated whenever an object is added to the database and are used by Database Services for various system tasks such as optimizing your database access.

There are nine types of Catalog Tables. Their names and functions follow.

| NAME | FUNCTION |
|---|---|
| SYSTABLES | Provides information on tables and views |
| SYSCOLUMNS | Provides information on the columns in tables |
| SYSINDEXES | Provides information on the indexes |
| SYSVIEWS | Provides information on views |
| SYSVIEWDEP | Provides information on view dependencies |
| SYSPLAN | Provides information about access plans |
| SYSPLANDEP | Provides information on access plan dependencies |

| | |
|---|---|
| SYSSECTION | Contains the sections of each access plan |
| SYSSTMT | Contains the original text of all embedded SQL statements |

You can use SELECT statements to reference data in the Catalog Tables in the same manner you reference data in your own tables. For the most part, the information in the Catalog Tables is meaningful to the application developer. For example, SYSTABLES contains a row for each table or view in the database and includes information such as the qualifier, the time created, and various aspects of the logical and physical structure of the table. The exact structure of the catalog tables can be found in the *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference*.

## Recovery Log

The log contains a running record of changes made to tables in a database. This log serves two purposes. First, it provides a mechanism by which transactions can remove changes that they have made. Second, the log provides a method of recovery from system failure.

The log supports transactions by recording what happened in any change to the database. Since an independent record of any database change is made in the log, the sequence of changes making up a transaction can be removed from the database if the transaction is rolled back.

In the case of a system failure, the database can be restored to a previous state by using information recorded in the log. In recovery, any changes made permanent with a COMMIT that have not been recorded in the database itself can be restored from the log. Any transactions that were in process when the system failed are rolled back. Logging is always active and cannot be turned off.

Another reason for keeping a log is to provide forward recovery from media failure. Forward recovery requires a previous backup version of the database and a log containing changes made to the database since that backup was made. This type of logging is sometimes referred to as archival logging.

The Database Manager log is a working log that is reset to a minimum size when there are no outstanding uncommitted changes to a database. This type of log cannot be used for forward recovery. If the volume containing your database becomes unusable, your only method of recovery is from a backup copy of that database.

## Database Configuration File

Like all computer software applications, the Database Manager uses resources when it is installed and when it runs. In the single-tasking environment of the PC DOS, configuration of resource management involves solving problems such as not filling up your hard disk and having enough room left in your 640K (after you have installed your favorite resident programs) to run your single, main application. In the OS/2 environment, resource management is more complicated than in PC DOS. Runtime resources under OS/2 (for example, RAM and shared control blocks) are managed by the operating system.

However, an application that is greedy for system resources can cause problems for itself and other concurrent applications.

Database Services acts as an extension to the operating system and provides a way for you to control the consumption of scarce systems resources used by all database applications. This control allows you to tailor the Database Manager for performance in different environments.

Database Manager maintains two sets of configuration parameters. One set of parameters exists for the Database Manager system itself and controls such resources as total shared RAM used by all database applications. A second set of parameters exists for each database and controls resources used by that database. Chapter 5, "Database Configuration Parameters," details each of the configuration parameters.

## DATABASE SECURITY

The Database Manager lets you protect a database with a password. After supplying a password for a database, you must use that password each time you connect to the database.

Passwords are of limited value since all data is stored on OS/2 volumes as files, and there is no way to prevent a user of the PC containing Database Manager databases from accessing any and all database files. Since these files are not encrypted, table data can be read, with some effort, by an editor. The database files can also be updated, which may destroy their internal structure and make them unusable by the Database Manager. Database files are vulnerable in this manner because there is no file security (or any security, for that matter) in Version 1.1 of OS/2.

Because of the lack of security in the OS/2 base operating system, a database password gives only casual security to Database Manager databases on a PC. If your database users are not familiar with the OS/2 file system and are having trouble finding and erasing the files, you may want to consider the use of database passwords. However, since the database password imposes a rather severe penalty on the legitimate user who forgets it, we recommend that you never assign a password to a database. Future releases of the Database Manager will use the SQL GRANT and REVOKE statements to provide access control to the objects in your database.

## CONCLUSION

You should now be able to choose and design the data objects you require such as tables, columns (data elements), indexes, and views. It is very important that you begin your design work at this level before actually writing your program or developing your Query Manager queries. You should also be aware of the role that the supporting objects play; these objects become more important in later design stages.

Chapter 4, "Database Consistency," continues the presentation of design information and explains the effect that data consistency features of the Database Manager may have on your application design.

# CHAPTER FOUR

# Database Consistency

## OVERVIEW

This chapter discusses the three mechanisms that the Database Manager uses to maintain the consistency of your data. These mechanisms are transactions, locking, and logging. The information in these detailed sections helps database and application designers understand the capabilities that a database system provides.

## WHAT IS CONSISTENCY?

As the owner of a database (relational or non-relational), you may find a set of changes to your database that you consider incorrect. These incorrect changes can affect both the integrity and consistency of your data. The integrity of your data can be compromised by malicious erasure of the database or access by unauthorized individuals.

Database consistency is best explained through an example. Suppose your database keeps track of parts in several warehouses. If some parts are transferred from one warehouse to another, the parts need to be removed from one table and added to another, thereby updating the database. If the application removes the parts from the first table but neglects to add them to the second, your database is then inconsistent because some parts are missing from the database. A database becomes inconsistent if the user running your application forgets to make all necessary changes (as in the example), the system crashes in the middle of making the changes, or the application stops prematurely for some reason. Also, if several users are on the system at once, a user might read your changes before the transfer is completed and take some inappropriate action or make an incorrect change to the database based upon the premature data they read.

Maintaining consistency requires solutions to the following:

- **How can the implementation maintain generic consistency of a database since you do not know what each individual database owner wants?**
- **How can you keep a single application from accidentally destroying database consistency?**
- **How can you ensure that multiple applications simultaneously accessing the same data will not destroy database consistency?**

- **If the system fails while a database is in use, how can you restore it to a consistent state?**

In the Database Manager, database consistency is maintained through the use of *transactions*, *locking*, and *logging*.

## TRANSACTION OVERVIEW

A transaction groups a sequence of SQL operations as a single unit. All changes within that transaction are either made permanent by a SQL COMMIT or are removed by a SQL ROLLBACK. An application can do all of its work as just one transaction, or its operations can be divided into a number of sequential transactions. Data either read or changed within a transaction is protected from being changed or seen by other transactions. Let's see how transactions answer the four questions posed in the first part of this chapter.

1. **How can the implementation maintain generic consistency of a database since you do not know what each individual database owner wants?** Transactions provide generic consistency by ensuring that changes made only by COMMITted transactions become permanent. If a transaction is backed out via a ROLLBACK, the data changes are removed. You use this generic consistency to enforce your application's consistency requirements.

2. **How can you keep a single application from accidentally destroying database consistency?** To solve this problem, you must ensure that the sequence of SQL operations in each transaction results in a consistent database. The Database Manager then ensures that each transaction is either completed as a unit (COMMIT) or removed (ROLLBACK). In this way, *your* definition of data consistency is maintained.

Using an application that does not understand your definition of consistency can destroy the consistency of your database. One way this can happen is if you use a database created by an application that groups SQL statements into units of consistency and later decide to use it with some generic user interface that ensures consistency by COMMITting after every SQL statement. For example, suppose you use such a generic interface in the warehouse scenario above. If you delete the parts from one warehouse but forget to insert them in the second, the warehouse database would be inconsistent. If consistency constraints have been defined on a database, you should only use applications designed to maintain that consistency.

The distinction should now be clear: it is the responsibility of the user or application to enforce the particular definition of database consistency chosen for that database through the use of transactions, and it is the responsibility of the Database Manager to keep the database consistent on transaction boundaries.

The level of interaction between transactions is called *data isolation*. The Database Manager uses a form of data isolation called *Repeatable Read*. Repeatable Read (RR) means that your transaction can read a row of data and that the data will not be changed by another transaction. Thus, the data can be read repeatedly and will always return the same value. Repeatable Read can be further defined with the following two rules:

- Rows read by your transaction cannot be changed by other transactions. These rows may be read by other transactions.
- Rows changed by your transaction cannot be read or changed by another transaction until your transaction ends.

The RR level restricts concurrent data access. Once data has been read by an inflight transaction, no other transaction can change the data until the first transaction ends. Transactions must be COMMITed frequently to allow adequate concurrency when reading and updating. Other database products, such as IBM's DB2, include additional forms of data isolation. Each form has its own set of rules that define which rows of data simultaneous transactions may access.

The foregoing has been a discussion of what transactions are rather than how they work in the Database Manager. A discussion of how transactions work is contained in the following sections on locking and logging. Locking and logging form the reality of the transaction environment. The cost of supporting transactions, how transactions interact, and what problems can occur are all covered.

## LOCKING OVERVIEW

3. **How can you ensure that multiple applications simultaneously accessing the same data will not destroy database consistency?** Locking solves this problem because the data is locked once it has been accessed within a transaction. The locks remain on the data until the transaction ends.

When a data object (table or row) in the database is accessed, that data object is locked. The lock prevents other transactions from accessing the data object in a way that would interfere with the original application. If a transaction is COMMITed, any changes to the data are made permanent as well as being unlocked for use by other applications. The locks are also removed when a ROLLBACK occurs. Locks are used to enforce the Repeatable Read level of data isolation. See "Transaction Overview" for more information on Repeatable Read.

If one transaction tries to access data in a way that is incompatible with a lock held by a second transaction, the first transaction must wait until the second transaction has ended. This is called a *lock wait*, and is familiar to users of a program such as Query Manager. When lock wait occurs, the screen simply "locks up" until the transaction holding the lock ends.

If you want to know more about locks, read "How Locking Works" in this chapter.

## LOGGING OVERVIEW

4. **If the system fails while a database is in use, how can you restore it to a consistent state?** If a database is active when a system failure occurs, that database is left in an inconsistent state until the next access. At that time, a special recovery process is run that restores the database to a consistent state. This consistent state is defined by the

transaction boundaries of any applications that were using the database when the system failed.

The recovery process is able to do its job since every change to a row is placed into a log file. Since the log contains both a "before" and "after" image of a changed row, any change can either be removed or restored to the database as necessary.

The log is also used during normal operation of the Database Manager. Should your transaction need to ROLLBACK its changes, the log is referenced in order to undo those changes.

The log can consume a considerable amount of disk space. For this reason, you can limit the log by specifying a minimum size and a size for extensions to acquire when the log gets full. You can also limit the size of the log so that it does not use all of your disk. For more detail on logging, see "How Logging Works" in this chapter.

## HOW LOCKING WORKS

This section discusses the details of locking and how it controls concurrent access to data. Since locks are implicitly obtained on behalf of your transaction, you need not explicitly request that an object be locked. Even so, there are some rules by which the Database Manager acquires locks. Understanding these rules might allow you to make better use of the system and increase the concurrency of your transaction.

### Types of Locks

As your transaction performs its operations, the Database Manager automatically acquires locks on the data objects it references. These implicit locks are either on a table or both a row and a table. The only object that you can explicitly lock is tables. You can only affect row locks with the COMMIT and ROLLBACK statements.

There are two primary kinds of locks:

| LOCK | FUNCTION |
|---|---|
| X lock | Gives exclusive use to the resource being locked. X locks are acquired for resources that you are going to UPDATE, DELETE, or INSERT. |
| S lock | Gives shared use of the resource being locked. An S lock is acquired to prevent someone else from UPDATEing what you have read. |

There are also special locks used only on tables. They are called *intention locks* and are used to signify that rows within the table are being locked. These locks are always placed on the table before any rows within the table are locked.

| LOCK | FUNCTION |
|---|---|
| IS lock | The table is locked with the intention of acquiring S locks on rows. This lock is acquired when your operation *does not* convey the intent to update any rows in the table. |

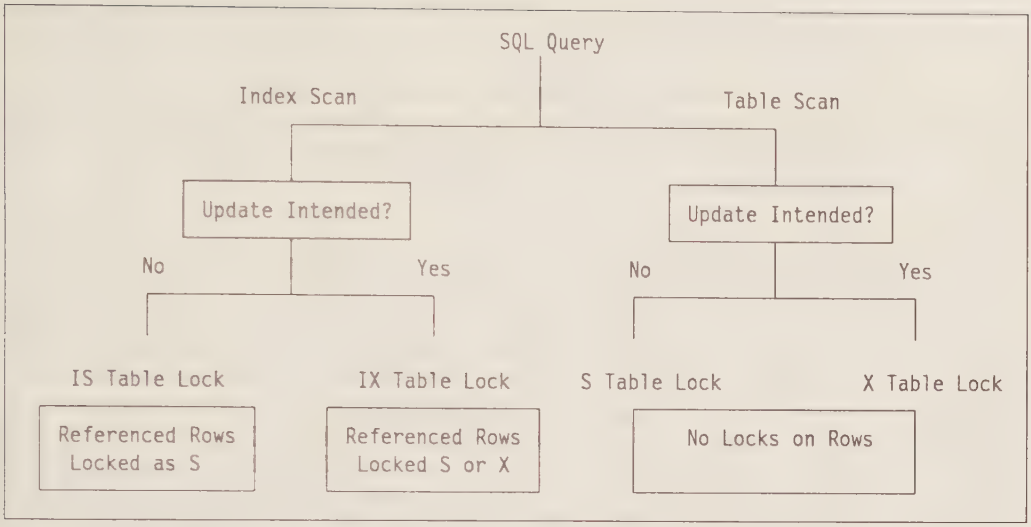



**Figure 4-1** How Lock Requests are Determined

IX lock — The table is locked with the intention of acquiring X locks on rows. This lock is acquired when your operation *does* convey the intent to update rows in the table. SELECTing FOR UPDATE, UPDATE WHERE, and INSERT all convey the intent to update.

## Lock Transitions for a Single Transaction

Let's begin by considering the locks acquired by a single transaction. When your transaction performs its operations, locks are acquired as objects are referenced. When you first reference an object, the Database Manager determines the type of lock that matches your request.

Whenever search arguments are specified on an SQL query (for example, a WHERE clause), then a scan of the appropriate table(s) is performed. A scan of a table is a search to find all the rows that match the search conditions (even if it is only one row). The type of scan, combined with the intent of the SQL query, determine the type of lock the Database Manager acquires on the table. Figure 4-1 represents the following process:

| SCAN | FUNCTION |
|---|---|
| table scan | If there are no indexes the optimizer can use, the table is searched row by row for the rows that match your search condition. Since tables are unordered, all rows are read. The table is locked either S or X depending upon the intent of the scan. Since all rows are read, no locks are acquired on the rows themselves. |

index scan — If there is an index the optimizer can use, the index is searched for keys that match your search condition. The rows associated with the matching index keys are directly accessed using the index pointers. If the SQL query conveys the intent to update, the table is locked IX and any referenced rows are locked X or S. If no intent to update can be determined, the table is locked IS and any referenced rows are locked S.

The presence of appropriate indexes on a table can increase concurrency. See "Effects of Locking on Concurrency" in this chapter for more information.

There is a hierarchical relationship between the lock on a table and the locks on rows within that table. When a table is locked, all the rows in the table are implicitly covered by the table lock. Conversely, when rows in a table are locked, the table must also be locked in an appropriate manner. The table lock prevents another transaction from locking the table in a manner incompatible with existing row level locks.

As your transaction continues processing, it may access an object for which it already owns a lock. This can happen when your transaction OPENs a second cursor or executes a DELETE, UPDATE, or INSERT for a row that has only been read before. Since only one lock can be held by a transaction at any time on an object, the single lock must cover all the requests made up to that point. The lock may have to be converted in order to satisfy this request. For example, if your transaction holds an IS table lock (SELECTed via index scan) and you OPEN a new CURSOR with a FOR UPDATE (SELECTed via index scan), the IS lock is converted to an IX. If your transaction already had an IX it would have covered the new request and no conversion would be necessary.

Locks acquired on rows can cause the table lock to be converted. For example, a table that is locked IS must only contain rows that are locked with an S. If the transaction wishes to acquire an X lock on a row, the IS lock at the table level is first converted to an IX. This is an example of the general rule that the table level lock must always "cover" the row level locks.

There is a hierarchy of locks that indicates which locks cover the others. This hierarchy also indicates the valid conversion paths that locks may take. Figure 4-2 is a representation of this hierarchy. Notice that an X lock covers all subsequent lock requests from a transaction on an object, but an IS lock only covers subsequent requests for IS locks.

You should notice that Figure 4-2 contains a lock type that we have not discussed, the SIX lock. The SIX lock means that the table is locked with an S lock with the intention to acquire X locks at the row level.

The SIX lock is always obtained as the result of a conversion. The S lock and the IX lock have an unusual relationship in that neither covers the other. If an S lock is held (table SELECTed for read) and an IX lock is required (intend to UPDATE), the conversion is made to an SIX lock. A SIX is held by a transaction that wants to S lock the whole table and intends to lock individual rows with X locks.

Lock conversion can only cause locks to move up the hierarchy. Locks stay at the highest level obtained, even if you CLOSE all CURSORs and reOPEN them. Only a COMMIT or ROLLBACK can remove the locks.

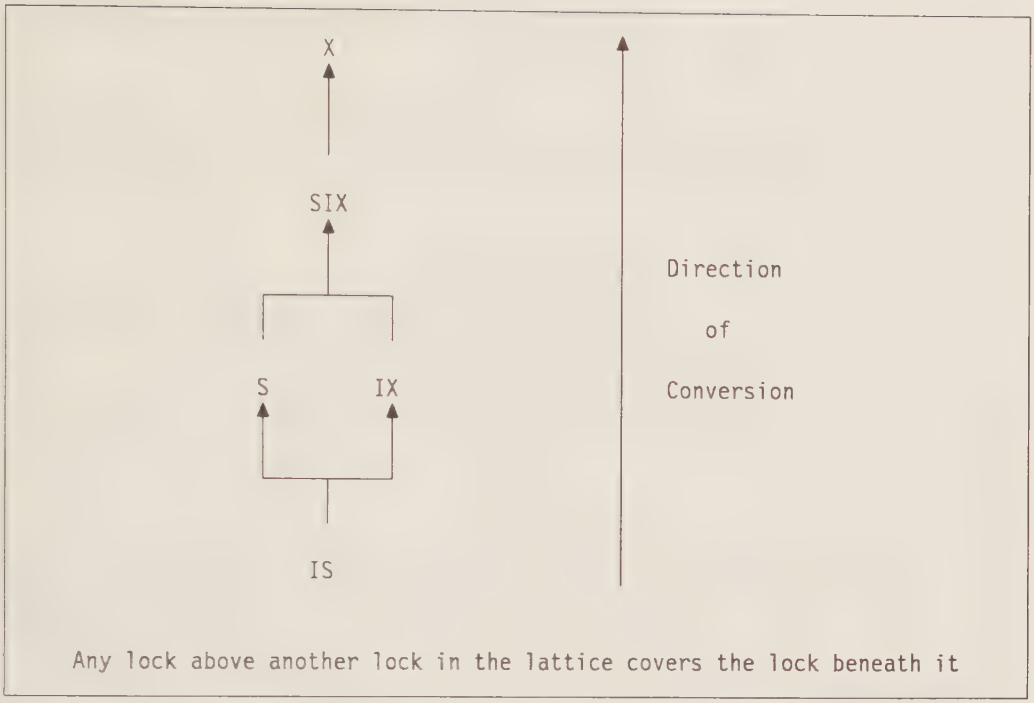



**Figure 4-2** The Hierarchy of Locks

### *Escalation*

Locks require space in the Database Manager. Since this space is not infinite, the Database Manager limits the amount of space that the system can use for locks. Furthermore, a limit is placed on the space *each transaction* can use for its own locks. These limits can be configured. See Chapter 5, "Database Configuration Parameters," for more information.

When an application using the database reaches one of these limits, a process called *lock escalation* occurs. Escalation is the process of converting locks on individual rows into a single table level lock. Space in the lock list data structure is then returned for use by other transactions.

In the escalation process, one of the transaction's tables is selected for escalation. If enough locks are not freed up, another table is selected for escalation, and so on, until enough locks have been freed to continue. If there is still not enough space in the lock list after all the transaction's tables have been escalated, the transaction is asked to either COMMIT or ROLLBACK (that is, it receives an SQL -912 error code).

The important point here is that an attempted escalation occurs to the process that encounters a limit. This result is expected if a "guilty" transaction reaches its own limit. However, if the system limit is reached, an "innocent" transaction (that is, one not holding many locks) might try to escalate, fail, and then be rolled back. Therefore, transactions

that misbehave by holding many locks over a long period of time can cause other transactions to COMMIT or ROLLBACK prematurely.

The effect of a successful escalation on a table is normally a reduction or elimination of concurrent access to that table by other transactions. Additionally, the escalating transaction may have to wait for other transactions holding incompatible table locks to COMMIT or ROLLBACK. The next sections explain further details on lock compatibility.

### Lock Compatibility Among Several Transactions

Your transaction might not be the only one acquiring locks on an object. This section discusses the compatibility between locks held by different transactions.

First, let's review the locks that can be held by a single transaction. On tables, a transaction can hold either an IS (Intention Share), IX (Intention Exclusive), S (Share), SIX (Share with Intention Exclusive), or X (Exclusive) lock.

On rows within a table, a transaction can hold multiple S (Share) and X (Exclusive) locks.

A transaction may hold one lock at a time on a table. Another transaction (or more than one) may hold a lock on the same table if the locks are compatible. The set of transactions concurrently accessing the same table is called a *granted group*. For example, any number of transactions can concurrently hold an S lock on a table. In this example the granted group can be large. On the other hand, only one transaction can hold an X lock on an table. The granted group here is just one transaction.

Figure 4-3 shows the compatibility matrix for locks held by different transactions. The matrix indicates that the following transactions are compatible:

- Multiple transactions holding IS locks
- Multiple transactions holding IX locks
- Multiple transactions holding IS and IX locks
- Multiple transactions holding IS locks and one holding an SIX lock
- Multiple transactions holding IS and S locks
- Multiple transactions holding S locks

An X lock is not in any group above because it is not compatible with any other lock. A transaction holding an X lock excludes all other transactions from accessing the locked object.

Granted groups can exist on rows also. Since the only locks available on rows are S and X, the compatibility matrix is much smaller (just the X and S columns and rows from Figure 4-3). Looking at that portion of the matrix, you can see that the only granted group for rows is multiple transactions holding S locks.

| Lock Held by First Transaction | Lock Requested by Second Transaction: IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | Yes | Yes | Yes | Yes | No |
| IX | Yes | Yes | No | No | No |
| S | Yes | No | Yes | No | No |
| SIX | Yes | No | No | No | No |
| X | No | No | No | No | No |

Yes - second transaction compatible

No - second transaction not compatible

**Figure 4-3** The Lock Compatibility Matrix

## Effects of Locking on Concurrency

By now it should be obvious that multiple transactions can concurrently access data if all they do is read. This fact is independent of whether indexes are defined or not. Concurrency becomes more of a concern when you introduce at least one transaction that is performing UPDATEs.

Unless an appropriate index is defined on a table, there is almost no concurrent write access to that table. Without an index, a table scan is performed; this means that the requesting transaction holds an S or an X lock on the table. Concurrent updates are possible only with the use of IS or IX locks, which are possible only if an index scan is performed.

Simply creating indexes on tables does not guarantee concurrency either. The Database Manager optimizer decides for you whether indexes are used in the processing of your SQL statement. So, even if you have indexes defined, the optimizer may choose to perform a table scan for any of several reasons:

1. **No index is defined for your seach criteria (WHERE clause).** The key must match the columns used in the WHERE clause so that the optimizer can use the index to help locate the desired rows.

2. **Direct access may be faster than via an index.** The table must be large enough so that the optimizer thinks it is worthwhile to take the extra step of going through the index rather than just searching all the rows in the table (for example, the optimizer would probably not use any index defined on a four row table).

3. **A large number of row locks would be acquired.**

Of this list, the first point is most important. If you choose to optimize for high concurrency, you should ensure that your table design includes a primary key for each table to be updated. Remember to use only this key when referencing your table for UPDATE. Taking these steps gives the optimizer the best opportunity to use the index you provide.

## Deadlocks

*Deadlock* can occur between two or more transactions waiting on locks. When a deadlock occurs, all the transactions involved in the deadlock wait indefinitely unless an outside agent takes action to end the deadlock. The Database Manager contains a process associated with each active data base that finds and resolves deadlocks in the locking subsystem. This process is called the deadlock detector.

An example of a deadlock between two transactions is as follows:

- Transaction T1 holds a lock on row R1.
- Transaction T2 holds a lock on row R2.
- T1 then requests a lock on R2 but must wait because T2 holds an incompatible lock on the same row.
- T2 then requests a lock on R1 but must wait because T2 holds an incompatible lock on the same row.

Because both T1 and T2 are waiting, they are "asleep" and can do nothing to resolve the fact that they are in deadlock.

It takes two or more transactions to cause a deadlock. A deadlock is more precisely referred to as a *deadlock cycle*, since the transactions involved in a deadlock form a circle of wait states. Each transaction in the circle is waiting for a lock held by one of the other members of the circle. The likelihood of a deadlock cycle containing more than two transactions is very low.

The example given at the beginning of this section showed two transactions in a deadlock on two separate objects (R1 and R2). Since a deadlock requires at least two transactions to make a cycle, you might assume that two data objects are always involved in the deadlock. This is not true, a deadlock can occur on just one data object. This deadlock condition is called a *conversion deadlock* because it occurs when a transaction is requesting a lock conversion.

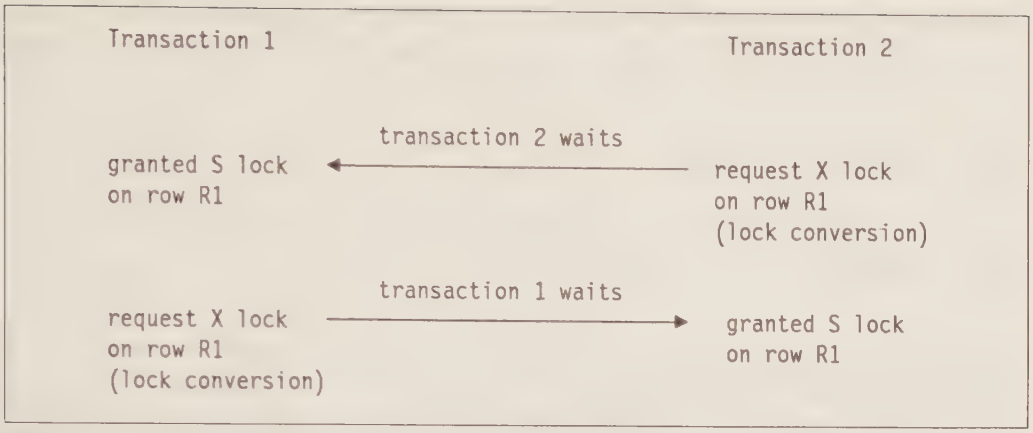



**Figure 4-4** A Conversion Deadlock in the Database Manager

A conversion deadlock happens when two or more transactions already hold compatible locks on an object and each then requests new, incompatible lock modes on that object. Figure 4-4 shows an example of such a deadlock.

A conversion deadlock is a common form of deadlock that usually occurs between two transactions performing index scans. Using an index scan, each transaction acquires S and X locks on rows. A conversion deadlock occurs when each transaction has read the same row and then each attempts to update that row. Applications designers need to watch out for conversion deadlock scenarios when designing high concurrency applications that are to be run by multiple, concurrent users. In situations where it is likely that the same set of rows may be read and then updated by multiple copies of the same program, that program should be designed to ROLLBACK and retry its operation in the event of a deadlock. As a general rule, the shorter your transaction, the less the likely it is to get into a deadlock.

### *Deadlock Detector*

When a database becomes active, the deadlock detector is started as part of the process that initializes the database for use. The deadlock detector stays asleep most of the time but wakes at preset intervals to check on whether any deadlocks exist between transactions in the database. The wakeup interval is determined by a database configuration file parameter. See Chapter 5, "Database Configuration Parameters," for more information on this parameter.

Normally, the deadlock detector wakes up, sees that there are no deadlocks on the database, and goes back to sleep. The deadlock detector uses very little runtime resources but is part of an active OS/2 process associated with a database.

If the deadlock detector discovers a cycle, it selects one of the transactions in the cycle as a victim. This transaction is then rolled back automatically. Your application receives the -911 SQL error code. When the ROLLBACK completes, all the victim's locks are released. The transaction waiting for the victim can now proceed since the deadlock cycle

is broken. It is possible but very unlikely that more than one deadlock cycle exists on a database. If this is true, the detector finds each remaining cycle and chooses another victim transaction to ROLLBACK until all deadlock cycles are broken.

## HOW LOGGING WORKS

In our discussion of log operations, we'll first look at what the log contains, log synchronization, managing log space, and finally talk about the recovery process.

### Log Records

Logging is simply a method of keeping track of what changes were made to a database. Every change to a row of data in the database is recorded in the log in an entity called a *log record*. A log record enables the Database Manager to either remove or restore the change. A change to a row in a table is either an UPDATE, an INSERT, or a DELETE. The log record may contain a "before" image and/or an "after" image of the row changed.

| | |
|---|---|
| INSERT | There is no before image, so only the after image of the new row is logged. |
| UPDATE | Both the before image and after image of the row changed is logged. |
| DELETE | There is no after image, so only the before image of the row is logged. |

These log records are the "data" log records, and they make up the bulk of the records in the log. There are other log records, but the only ones of interest here are those that indicate a COMMIT or ROLLBACK has occurred. A COMMIT or ROLLBACK record ends a sequence of data log records for a transaction. Whenever a COMMIT or ROLLBACK log record is written, it is immediately forced out to the disk log. This ensures that all the log records of a completed transaction are on the disk log and cannot be lost due to a system failure. This is another way to define a COMMIT or ROLLBACK.

Because more than one transaction may be using a database at a given time, the log contains the changes made by multiple transactions. To keep everything straight in the log, each log record contains an identification of the transaction that created it. In addition, all the log records for a transaction are chained together.

If a COMMIT occurs, the log records are no longer needed. If a ROLLBACK occurs, the Database Manager processes each log record written by the transaction in reverse order and backs out all changes. This can be done because the before image of the rows are contained in the log.

### Log and Database Synchronization

Database Services can maintain consistency only by keeping the log and the database in synch. This synchronization is achieved through a *write ahead logging* technique.

To understand this consistency problem, let's discuss how the disk database gets up-

dated. When you change a row in a table, that change is actually made in a RAM buffer (contained in the database buffer pool). The changed row is written to the disk later. A buffer containing a change that has not been written out to disk is said to be "dirty."

The most current data changes to a working database are in the buffer pool but not out on the disk. Write ahead logging preserves consistency by forcing the log record of a change out to the disk before the dirty buffer containing the changed row is written to disk. Log records are forced to disk when

1. A transaction performs a COMMIT or ROLLBACK. All log records up to and including the COMMIT record are forced to the disk log.
2. The buffer pool manager is ready to write a dirty data buffer. All log records affecting that buffer must be in the disk log. All log records including the last one to affect the buffer are written out.

If the system crashes, the database on disk may contain changes made by uncommitted transactions (making the data base inconsistent). In this case, the disk log contains a record of every uncommitted change made to the database. We know this because the log record of the change must be on disk before the actual change is written. This allows the recovery process to restore the database to a consistent state. The recovery process is discussed in more detail in "Recovery" later in this chapter.

### Managing Log Space

Now that we have discussed what the log contains and how it is used, we can explore the topic of how log space is managed. As you have seen, Database Services writes records to the log sequentially to support transactions. If this were to continue, the disk would eventually become full with log records. The Log Manager controls the size of the log and when possible resets the log to an empty state.

The log is called a linear log because it grows until it is reset. The growth of the log is governed by three database configuration parameters: *minimum size of the log*, *size of an extent*, and *number of log extents*. The meaning of these parameters and how they are set is covered in "Log Manager Parameters," in Chapter 5.

When the log becomes empty due to transaction inactivity, the log extents are released. Inactivity means that no transaction has any records in the log. A transaction that runs out of log space, either because the maximum size was reached or because there was too little disk space to allocate the next extension, a ROLLBACK occurs. Regardless of cause, this continues until the log inactive state is reached and the log is reset to its minimum size. The two scenarios that prevent the log from being reset are continuously overlapping transactions and lengthy transactions.

Continuously overlapping transactions are not likely but can happen when transactions starting at close intervals use a database (that is, high volume or high activity rate transactions). If you are designing a database system in which the transaction arrival rate is high, you should increase the size of the log. This increase reduces the probability of your transactions being rolled back due to insufficient log space.

A lengthy transaction, in this case, is one that causes many log records to be written before its COMMIT occurs. How these applications are used must be considered as well as the amount of log space needed to support them. Any other transactions running concurrently with a lengthy transaction tend to aggravate the log space requirement. An application that runs by itself (no other transactions) should probably open the database for exclusive usage and then fill up the log before making its COMMIT. An application that never ends execution and never does a COMMIT is a faulty application design since it eventually causes itself (and possibly other transactions) to fail.

### Recovery

Recovery is the process of returning the data in a database to a consistent state in the event of a system failure. A simple example of such a system failure is turning off the power in the middle of a work session. Recovery in the Database Manager is based on the database log.

Database Manager determines recovery is needed the first time the database is opened after the system failure occurs. This is accomplished by looking at the log and determining that the database was not shut down normally. In such a case, the log file may contain records that were not written to the database. These changes were still in the RAM data buffers when operations were abnormally terminated. The disk image of the database could be inconsistent since changes made by completed transactions may have been lost. However, all changes to the database were kept in the log file, which is not buffered in RAM. The changes kept in the log file are used to restore the database to a consistent state.

Let's assume that the database is in an unknown state. It may contain changes made by uncommitted transactions. These changes are written to the database when the RAM data buffers get full and the least recently used one is forced to disk. The database may also be missing changes that were made by committed transactions. These changes were still in the RAM data buffers when the problem occurred but were recorded in the log. To restore the database to a consistent state, the Database Manager does the following:

1. Any change made by a transaction that was inflight (had not committed or rolled back) is removed from the database. The Database Manager works backward through the log and if an uncommitted change is found in the database, the record is restored to the before image retrieved from the log.

2. Any change made by a committed transaction that is not in the database is written to the database. As the Database Manager scans the log, any committed log record found that is not in the database will be written to the database.

3. If a transaction was in the process of a ROLLBACK, the ROLLBACK is completed so that all changes made to the database by that transaction are removed.

Although everyone agrees that restoring committed changes to the database is a good idea, some may wonder why the inflight changes are removed. The answer is that the

Database Manager knows that changes are consistent only when a COMMIT is explicitly issued by a transaction. Therefore, all the work done by the inflight transactions are considered inconsistent and must be backed out of the database to preserve database consistency.

### *Soft Checkpoint*

As described above, during the recovery process the Database Manager must scan the log to restore the database to a consistent state. While scanning the log, the database is read to determine whether it contains the committed or uncommitted changes. If the log is large, the recovery process could take quite a while to scan the whole log and read associated rows from the database. Fortunately, it's usually unnecessary to scan the whole log since the actions recorded at the beginning of the log are "old." Let's review what old actions mean.

Old log records have been in the log longer than the others. Since they have been there longer, the chance is greater that their transactions have completed and that the data has been written to the database. Thus, no recovery actions are required for the log records generated by this transaction.

If there were some way to skip these log records during the recovery process, the length of time necessary to recover the entire database could be shortened. This is the purpose of the *soft checkpoint*. A soft checkpoint establishes a pointer in the log at which to begin recovery. All log records recorded earlier than this checkpoint are the result of completed transactions and the RAM data buffers containing the changes have been written to the database.

There is a parameter in the database configuration file that controls the frequency with which soft checkpoints are taken (see Chapter 5, "Database Configuration Parameters"). The overhead of a soft checkpoint is low, but it does require an additional I/O to update the pointer. A soft checkpoint is most useful when logs are large since it can reduce the number of log records examined during recovery. The more often a soft checkpoint is taken, the less work is needed to recover from a system failure.

## CONCLUSION

A good understanding of database consistency is extremely important before designing your database application. It is also one of the more complicated aspects of application design. As you build your application, you should reread portions of this chapter. You will find that it will help you catch potential problems with your application design.

The next chapter completes the information on database application design considerations by discussing the role of the database configuration.

# CHAPTER FIVE

# Database Configuration Parameters

## OVERVIEW

Database configuration parameters can affect whether your application runs and, in some cases, how well it runs. Several different parameters affect Database Services. Figure 5-1 depicts the relationship between the two types of parameters. The Database Services configuration parameters apply to all databases on your PC. In addition, each database has its own additional set of configuration parameters.

You can set these parameters with your application program at installation of the application. This may be acceptable if your application is the only application that uses a database as in the case of "vertical" applications. Vertical applications usually bundle both the hardware and software together and the software provides all the application functions needed. An example of a vertical application is having one or more PCs with software to run a dentist's office. Applications might include patient scheduling, text processing, billing, and inventory.

If you do not have a vertical application, you may want to document a suggestion for the configuration parameter settings or how to change them. Recommendations on parameter settings are made in this chapter. You can use these or determine what is best for your application.

You can provide a program to modify the configuration parameters although this may be more work than it is worth since the Query Manager provides this function already. This chapter describes the configuration parameters and how they can be set using the Query Manager. Both the Query Manager menu item name and Database Services API parameter names are provided. The latter name, since it is shorter, is used in making parameter references.

Query Manager menu references are made in the material that follows to describe how you would modify the database parameters. You should read Chapter 13, "Query Manager Usage and Terminology," before using the Query Manager to modify the parameters.

## DATABASE SERVICES CONFIGURATION PARAMETERS

The Database Services Configuration Parameters are accessed when Database Services is started (STARTDBM) and are used to make various decisions during initialization. Do not confuse these parameters with the "Database Configuration Parameters." The two Database Services Configuration parameters are **numdb** and **sqlnseg.**

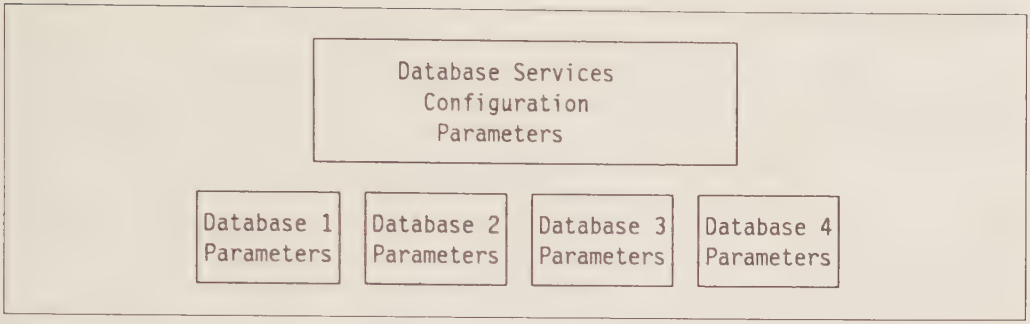



**Figure 5-1** Configuration Parameters

The first parameter, **numdb** (number of concurrent active databases), indicates the maximum number of databases that can be open concurrently. The same database can be opened by every OS/2 process without any problems. A "START USING database-name" request to Database Services causes a database to be opened. Database Services will fail (return an error return code) all requests to open an additional data base above this number.

The value of **numdb** can be set as follows:

**minimum** 1
**maximum** 8
**default** 1

The value picked for this parameter affects the value that must be specified for the **sqlnseg** parameter. If you need to change the value of this parameter, set it to 8 and set the value of **sqlnseg** to 41. You can avoid later resets if you set the value to the maximum.

The second parameter, **sqlnseg** (maximum segments), limits the number of OS/2 shared memory segments acquired by Database Services. In other words, this is the maximum number of OS/2 shared memory segments that can be acquired before Database Services fails the next distinct database that is opened. OS/2 shared memory segments are a limited systemwide resource.

As each distinct database is opened, a number of OS/2 shared memory segments are allocated. The number of segments acquired depends on the Database Configuration Parameters. These parameters are discussed in the next topic, "Setting sqlnseg."

The value of **sqlnseg** can be set as follows:

**minimum** Must be greater than or equal to

```
5 * (number of concurrent active databases) + 1
```

Therefore, the absolute minimum is 6. This value increases by 5 for each increment in **numdb.**
**maximum** 802
**default** 6

### Setting sqlnseg

If you increase the **locklist, buffpage,** or **dbheap** Database Configuration parameters, you may have to increase **sqlnseg.** Since this is a limit value, you should consider setting it high and not worry about it. If this seems wasteful, remember that a fully configured PC might have four fixed disks. If each has a single database and all databases are concurrently used, the minimum value needed for **sqlnseg** is 21. Since you may increase one of the Database Configuration Parameters affecting **sqlnseg,** you could set it to a value of 41 and probably not have any problems. If you are worried, note that 41 is less than 1% of the OS/2 shared memory segments.

It is unnecessary to change **sqlnseg** until you need to increase **numdb** or change one of the related Database Configuration Parameters. If you do change it, you should probably increase **numdb** to a value of 8, thereby avoiding any further changes.

If you want to be more precise in determining **sqlnseg,** take the following steps.

1. For each database on your PC, determine the values for the **locklist, buffpage,** and **dbheap** parameters. You can do this by selecting Tools on the action bar of Query Manager Databases Main Menu (Figure 13-2). You will see a pull-down menu that contains the *Reconfigure database* action. Selecting this action will display the current settings of the **locklist, buffpage,** and **dbheap** parameters for that database.
   Make the following calculation:

   ```
   segs = 2 + dbheap + ((locklist + 15)/16) +
   ((buffpage + 15)/16)
   ```

2. Total the values computed for segs (one per database), add 1 to that, then use this value for **sqlnseg.** You really only need to compute segs for those databases that are used concurrently.

To modify the Database Services Configuration parameters using the Query Manager, start the Query Manager and select System on the action bar of Query Manager Databases Main Menu (Figure 13-2). You will see a pull-down menu that contains two options used to alter the Database Services configuration parameters. These options are the *Reconfigure system* and *Restore system defaults.*

The *Reconfigure system* option allows you to set the Database Services configuration parameters. Since the Query Manager does not check your input parameters for consistency with respect to the parameter interdependencies, it will try whatever you enter. Query Manager then reports all errors from Database Services. Review the previous information provided on minimum values for the configuration parameters.

The *Restore system defaults* option restores the Database Services configuration parameter default values as provided by IBM.

Query Manager attempts to stop and restart Database Services for you so that changes you have made to the parameters will be put into effect. If another OS/2 task is currently using a database, Query Manager cannot stop the Database Manager since this would

probably disrupt other program(s). In this case, you must manually STOPDBM when the other OS/2 task(s) has completed. Then issue a STARTDBM command. Database Services prevents any subsequent database connections once its configuration file has been modified. The parameters will not be used until a STOPDBM and STARTDBM have occurred.

## DATABASE CONFIGURATION PARAMETERS

The Database Configuration Parameters govern the resource use for a particular database when it is opened. A list of the parameters, divided into the four areas of Database Services they affect, is shown below. *Shared memory* indicates storage in an OS/2 shared memory segment.

Lock Manager:

**locklist** Maximum storage for lock lists (# of 4K pages) shared memory

**dlchktime** Time interval for checking deadlock (# of seconds)

Buffer Pool Manager:

**buffpage** Buffer pool size (# of 4K pages) shared memory

**maxfilop** Maximum number of database files open per application

**maxtotfilop** Maximum number of total files open per application

Log Manager:

**logfile** Initial log file size (# of 4K pages) of disk storage

**softmax** Log records to write before taking a soft checkpoint

**logext** Size of one log file extent (# of 4K pages)

**logmaxext** Maximum number of extents to the log

Storage Manager:

**maxappls** Maximum number of active applications using this database

**applheapsz** Default application heap (# of segments)

**dbheap** Database heap (# of segments) shared memory

**sortheap** Sort list heap (# of segments)

Although this information is not stored in the database as are the tables, it is still associated with a particular database and there are APIs provided by Database Services to access and alter these parameters. You can write your own program to alter these values using the Database Services APIs. Do not confuse these parameters with the "Database Services Configuration Parameters." The Database Configuration parameters will be discussed in greater detail in the following subtopics.

### Lock Manager Parameters

The Lock Manager Parameters deal with concurrency between applications using the same tables in the database. Further information on this topic can be found in Chapter 4, "Database Consistency." The two Lock Manager parameters are **locklist** and **dlchktime.**

The **locklist** parameter indicates the number of 4K pages allocated to hold the lock list. This parameter should be large enough to allow all application transaction workloads to acquire their locks. If there is insufficient lock space, the applications will receive a -912 SQL error code (out of lock space) and they will be rolled back.

The value of **locklist** can be set as follows:

**minimum** 4
**maximum** 250
**default** 8

Concurrency and the number of locks needed are extremely difficult to predict. Lock list entries exist only for the duration of a transaction and should not tie up storage for very long. The default value should be adequate. If necessary, increase the default setting by 2 until there are no more -912 error codes. If -912 errors persist, attempt to determine the applications being run. They may not be COMMITing their transactions often enough. This application's usage of the database can also lead to other transactions being locked out for long periods of time.

The pages are suballocated in OS/2 shared memory segments and thus affect the **sqlnseg** parameter. If you change the **locklist** value, review "Setting sqlnseg" in this chapter to determine if other changes are required.

The **dlchktime** parameter indicates the interval of time between checks for deadlocks in the set of locks being waited for. A deadlock condition is undesirable because the work an application did before it entered the wait is rolled back and presumably must be repeated. Many applications put this burden back on the user to repeat commands and the inputting of data. Changes to this parameter have no impact on preventing deadlocks.

The value of **dlchktime** (in units of seconds) can be set as follows:

**minimum** 1
**maximum** 600 (10 minutes)
**default** 10

You can determine how quickly you learn about the bad news by changing this parameter. If you have a nice application that keeps its own journal and can automatically recover from deadlock, you may reduce the time interval. In an environment with little or no concurrency, you should set **dlchktime** high. In either case, the overhead in checking does not significantly impact performance.

### Buffer Pool Manager Parameters

The Buffer Pool Manager parameters deal with the interfaces between the OS/2 File System and Database Services. The Buffer Pool Manager parameters are **buffpage, maxfilop,** and **maxtotfilop.**

The **buffpage** parameter, probably the most significant parameter with respect to performance, indicates the number of 4K pages used by Database Services to hold disk data for this database. The buffer pool formed by these pages is not shared with any other database. The pool is separate from the buffer pool provided by OS/2 via the *Buffers* parameter in CONFIG.SYS. Data is written in a manner that prevents it from being saved in the OS/2 buffer pool. This avoids duplicate caching (saving the data in two different buffer pools).

The value of **buffpage** (in number of 4K units) can be set as follows:

**minimum** 2
**maximum** 1500 (6000K)
**default** 16 (64K)

The value for **maxappls** must not exceed half the value of the **buffpage** parameter. If you increase the value of **maxappls**, make sure your value for **buffpage** is at least double that value. The buffer pool pages are suballocated in OS/2 shared memory segments and thus affect the **sqlnseg** parameter. If you change this value, review "Setting sqlnseg" in this chapter to determine if other changes are required.

The default will probably be low for most users. The best way to judge whether this parameter is set correctly is to devise a repeatable test measuring the time it takes to perform some set of tests. By increasing (or decreasing) the **buffpage** parameter and measuring the effect, you will determine what its best setting is for your environment. Most buffer pools provide improved performance by reducing the need to do disk I/O to reread data previously read. An infinite buffer pool will not help performance of applications that only cause data to be written to the disk.

The **maxfilop** parameter limits the maximum number of OS/2 files concurrently open for use by Database Services running on an application's OS/2 process thread. It is normally set to a value less than **maxtofilop** to allow the application to open its own files while accessing the database.

The value of **maxfilop** can be set as follows:

**minimum** 2
**maximum** 235
**default** 20

If the Buffer Pool Manager hits the OS/2 limit (**maxtofilop**, unless the application changed it subsequent to starting the database), some open database files must be closed in order to open new ones. Opening and closing files by the Buffer Pool Manager is overhead that reduces performance. Occasional activity of this kind is normal, but opening and closing files whenever a new file is accessed is excessive. Depending on the number of files opened by your application, increasing this value may help performance.

The **maxtotfilop** parameter limits the total number of files that can be opened by both the Buffer Pool Manager and your application in a single OS/2 process. When an application starts a database, the current value of **maxtotfilop** is sent by the Buffer Pool Manager to OS/2 on a **SetMaxFiles** call. This resets the previous open file limit that OS/2 enforced

for that application (OS/2 process). In this way, the Buffer Pool Manager ensures that files can be open to the application and itself.

The value of **maxtotfilop** can be set as follows:

**minimum** 25
**maximum** 255
**default** 255

The default is equal to the maximum; little is gained in reducing it.

### Log Manager Parameters

The Log Manager parameters deal with the allocation of disk space for the transaction log. The transaction log is discussed in Chapter 4, "Database Consistency." It keeps a journal that can be used to back out changes made but not committed to the database. The primary consideration is that the log can grow large enough to handle a demanding sequence of overlapping transactions without running out of space.

The total size of the log is dictated by the first three parameters that follow. When the log needs to exceed the minimum size (**logfile**), an extent is acquired if the value of **logmaxext** is not zero. In this case, the first log extent (**logext**) is allocated and formatted. Then, if the log grows to need space in excess of what the first extent provides, a second extent is allocated and formatted. Assuming space is available from the disk, Database Services continues to allocate extents until it reaches the maximum number of extents (**logmaxext**).

If the log becomes full, the operation needing more log space is failed. In this case, that transaction can retry the operation, COMMIT, or ROLLBACK. The latter two actions will free up some log space and may allow other transactions to continue. When the log becomes empty (due to transaction inactivity), the **logext** areas are released.

There is a performance impact to acquire, format, and free log extents. Therefore, the **logfile** parameter should be large enough to accommodate transaction peak loads to avoid this overhead. Long running transactions will use up log space and may cause extents to be taken. You may want to increase the size of the log or the size of the log extents if you have long running applications that do not COMMIT their work very often. The Import utility is a long running transaction that does a COMMIT only when it completes or runs out of log space. Increasing the size of the log or having log extents for Import is probably not a good idea since it COMMITs when the log gets full.

The **logfile** parameter determines the minimum amount of disk space to be used for the database's recovery log. This amount of space is allocated when the database is created so that the applications using it are guaranteed a recovery log of at least this size. This parameter, in conjunction with the other parameters (**logext** and **logmaxext**), determines the total amount of recovery log space available.

The value of **logfile** (in number of 4K units) can be set as follows:

**minimum** 12(48K)
**maximum** 9999 (4 Meg)
**default** 50(200K)

This value can be set as high as 4,194 megabytes using the Database Services API. This is also the maximum size the log file can be, irrespective of the values for **logext** and **logmaxext.** This is considerably larger than the technology provided by most fixed disks. Optical disks (CD-ROM technology) extend the capability of storage media by a considerable amount.

The **logext** parameter governs the size of a log extent that will be requested from OS/2 when more disk space is needed for the log after **logfile** is full and any preceding log extents are full. When the applications using the database COMMIT their work, an attempt is made to delete the log extents. Therefore, log extents are only allocated and used as necessary.

The value of **logext** (in number of 4K units) can be set as follows:

**minimum** 4 (16K)
**maximum** 256 (1024K)
**default** 10 (40K)

The **logmaxext** parameter determines the maximum number of log file extents that will be acquired before a log full condition occurs. The value of **logmaxext** can be set as follows:

**minimum** 0
**maximum** 65535
**default** 20

This value can be set as high as 2,000,000 using the Database Services API. However, the total size of the log can not exceed 4,194 megabytes.

The **softmax** parameter is the number of log writes (individual log records written to the log) that occur between soft checkpoints. A soft checkpoint establishes a place in the log from which to begin recovery so that the Database Manager can skip over the first part of the log. The soft checkpoint is most useful when many overlapping transactions cause the log to grow large.

The value of **softmax** can be set as follows:

**minimum** 0
**maximum** 65535
**default** 100

The overhead of a soft checkpoint is limited to one additional log I/O to the disk (an updated log file header record is forced out to disk). The soft checkpoint information is used only to make a database recovery from the log more efficient.

### Storage Manager Parameters

The Storage Manager parameters deal with the allocation of memory (private or shared) for various components of Database Services. The Storage Manager parameters are **maxappls, applheapsz, dbheap** and **sortheap.**

The **maxappls** parameter indicates the maximum number of applications (separate OS/2 processes) that can concurrently use this database. The value of **maxappls** can be set as follows:

**minimum** 1
**maximum** 117
**default** 3

This value should be increased if you have several concurrent applications that will use this database. Ordinarily, you should set this parameter to 8. Changing this value may require you to increase **buffpage.** Review the parameter requirements discussed under **buffpage.**

The **applheapsz** parameter specifies the maximum number of segments of private memory available to Database Services when processing application requests. The value of **applheapsz** can be set as follows:

**minimum** 2
**maximum** 20
**default** 3

You usually will not need to modify this parameter setting. If an application receives a -954 SQL error code (out of application heap space), then increase the setting of this parameter.

The **dbheap** parameter specifies the number of shared memory segments available to process database requests. The value of **dbheap** can be set as follows:

**minimum** 1
**maximum** 45
**default** 1

You usually will not need to modify this parameter setting. If an application receives a -956 SQL error code (out of **dbheap** space), then increase the setting of this parameter. These segments are allocated as OS/2 shared memory segments and therefore affect the **sqlnseg** parameter. If you change this value, review "Setting sqlnseg" in this chapter to determine if other changes are required.

The **sortheap** parameter controls the amount of workspace (in segments) available to Database Services when sorting is required. The value of **sortheap** can be set as follows.

**minimum** 1
**maximum** 20
**default** 2

The default value supplied should be adequate for most applications. If you use SQL SELECTs (whose result table has many rows) and the SELECT has an ORDER BY clause, it may fail (SQL error code is -955) due to insufficient work space needed to sort (ORDER) the result. In this case, you will need to increase the **sortheap** parameter.

### How to Change the Database Configuration Parameters

To modify the Database Configuration parameters using the Query Manager, start the Query Manager and then select Tools on the action bar of Query Manager Databases Main Menu. The pull-down menu that appears contains two options. The *Reconfigure database* option allows you to set the database configuration parameters. The *Restore database defaults* option restores the database configuration parameters that came with the Database Manager.

## CONCLUSION

In this chapter you learned about the Database Services and Database configuration parameters and how they can be changed. These interrelated parameters determine the amount of resources allocated by Database Services to handle application requests. They can affect your application's performance and its ability to run successfully. You should consider the impact of changing these parameters when your application shares data or a database with other applications.

You should set or recommend setting the Database Services configuration parameters to high values, as discussed in the early sections of this chapter, since they are limit parameters. The resources controlled by the Database Services configuration parameters are allocated only as needed. The parameters prevent (limit) Database Services from using more than a defined aggregate amount of OS/2 system resources for all simultaneously opened databases.

The Database configuration parameters determine the amount of resources allocated when a database is initially opened, the amount of additional resources needed during operations, and some resource limit parameters. Their settings can affect performance. They may have to be changed to avoid SQL error conditions due to insufficient resources to run your application. More lock, buffer, log, and storage resources are usually needed when you have concurrent database applications using the same data tables.

The next chapter explains how Database Services uses OS/2 to store your databases and how this applies when you create a database.

# CHAPTER SIX

# Database Manager and OS/2

## OVERVIEW

You should think of Database Services as an extension of the OS/2 operating system. The services provided by Database Services can be used in a similar fashion to OS/2 system services, although Database Services also uses services provided by OS/2 in order to do its own work. You should understand how Database Services interacts with OS/2 before you design your databases and applications.

When you create a database, you specify a drive letter (A, B, C, and so on) to indicate which OS/2 volume will contain the data. Database Services stores your database on this volume as a set of OS/2 files. These files are kept in a directory named and created by Database Services. This chapter discusses how Database Services manages these files and directories, using OS/2 services for these and other tasks.

## DATABASE FILE LAYOUT

Database directories are created as children of an OS/2 volume's root directory. For each new database, a new OS/2 directory is created. To ensure that the directory name is unique, Database Services chooses the directory name. These directories are named starting with SQL00001. The numeric suffix is incremented for each new database. You have no control over the name of the directory or its placement on the volume. This volume can contain other files and directories without interfering with database operation.

Each database directory (C:\SQL00001, for example) contains several types of OS/2 files. These files contain the database objects discussed in Chapter 3. All file names are chosen by the Database Manager and have no bearing on the actual name of the object (table, index, etc.) represented. All files are stored in internal formats and should not be used with other programs. In other words, do not attempt to use your text editor on any of the files in the database directory.

Each table is stored in a single file with an extension of .DAT. If a table has LONG VARCHAR columns, there will be an additional file with an extension of .LF (Long Field). All indexes for a table are in a single file with an extension of .INX. Since each database has its own log file, you will find it in the database directory with an extension of .LOG.

You can request that a database be created on any OS/2 volume, including diskette volumes. When creating a diskette database, be aware of the initial size of a database. Since a newly created database contains the Catalog Tables and a log file, a good portion

of a diskette is already used up (approximately 512K bytes, using system defaults). Considering that the highest capacity diskette on a PS/2 holds about two megabytes of data, you are not going to be able to squeeze much of a database on a diskette.

### How Many Databases?

Although you can create as many databases as you wish, disk space is not unlimited. To answer the question of how many databases one should have, you need to understand several limitations:

1. **A database can exist on one and only one OS/2 disk volume.**

2. **Each database has its own recovery log.** The log initially requires 200K bytes of disk space (system default). This log can grow from its initial size due to concurrent application usage or a single application that does infrequent COMMITs. The log is returned to its original size only when all applications reach a state where they have COMMITed their updates and not yet begun any new changes. See "How Logging Works" in Chapter 4 for more information on the log.

3. **Only one database can be open per application process at a time.** This has the following implications:

   - Tables in different databases cannot be the object of a SQL statement (for example, you can not SELECT (join, etc.) information from tables in different databases). Tables must be in the same database.
   - Query Manager objects in different databases cannot be shared. The only possible technique is to copy the Query Manager objects from one database to the other databases.

4. **There is no limit to the size of a database.** While an individual row is limited to 4005 bytes, there is no limit to the number of rows that a table can contain. As a table grows larger, the Database Manager requests more free space from the OS/2 file system. One or more tables in a database can grow until there is no more space on the drive containing the database. The limit on the maximum space a file system can contain depends on the OS/2 base operating system. In practical terms, a table can grow almost as large as the largest file supported by the OS/2 volume. Conversely, there is no support in the Database Manager to limit the size of a database or a database table. A user or database administrator who needs to constrain the space used in a database must solve this problem outside of the Database Manager. A program could be written to monitor the free space left on the volume containing the database or the space required by the files in the database directory. Such a program could issue a warning when free space dropped to 10% of the total volume space or the database directory reached a prescribed upper limit.

As a general rule, you should have a single database per OS/2 volume. Instances where this is not feasible are listed below.

1. **Table naming conflicts** If you have different applications that reference their tables using specific names, a naming conflict can occur that would require having separate databases for each application. Before taking this step, determine if you can avoid duplicate names by using a different qualifier for the table name. If all else fails, contact the application provider for suggestions.

2. **Software developers** If you develop database application software, you may want to have separate databases to hold the same tables to do version testing of your software.

3. **Temporary situations** You may decide to create a data base temporarily. For example, when you initially install Query Manager you may want to use the provided SAMPLE database in learning to use the Query Manager. Later, you can erase the SAMPLE database.

### Volume and System Directories

In addition to the database directories, Database Services creates one additional OS/2 directory for each volume containing databases. This directory, SQLDBDIR, contains the Database Manager Volume Directory. Database Services uses the Volume Directory to locate the databases contained on that same OS/2 volume. The Volume Directory is represented by the box in the lower left corner of Figure 6-1. An entry for that database is automatically added to the Volume Directory whenever you create a database.

The volume on which the Database Manager code is installed contains some additional OS/2 directories. The first directory is the program directory itself, SQLLIB. The SQLLIB directory contains the following types of files:

1. **Database Manager runtime support** These files include the executable (.EXE) and command files (.CMD) used to start and stop the Database Manager. The dynamic link libraries (.DLL) and bind files (.BND) for the Query Manager and Database Services utilities are also in this directory.

2. **Program development support** These files include the C language include files (.H), link libraries (.LIB), and the SQL Precompiler and Binder.

3. **Query Manager profiles** All Query Manager profiles you create are placed in \SQLLIB.

4. **Query Manager import/export message logs** When you use the Query Manager for data import/export, any messages are recorded in a file stored in \SQLLIB.

Figure 6-2 shows the OS/2 directory structure of the disk on which the Database Manager is installed. Notice that there are two additional directories underneath \SQLLIB. The directory indicated by the dashed line, \SQLLIB\QMF, is created when you run the SQLQMF command to get a table from DB2 or SQL/DS. See "Importing Data from DB2 or SQL/DS" in Chapter 16 for more information on this utility. The other directory, \SQLLIB\SQLDBDIR, contains the Database Manager System Directory.

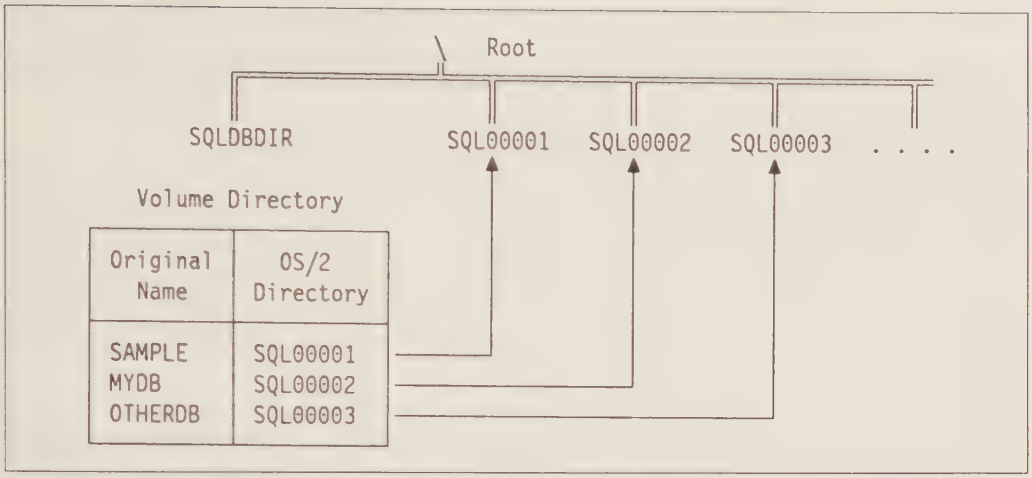



**Figure 6-1** An OS/2 Volume Containing Databases

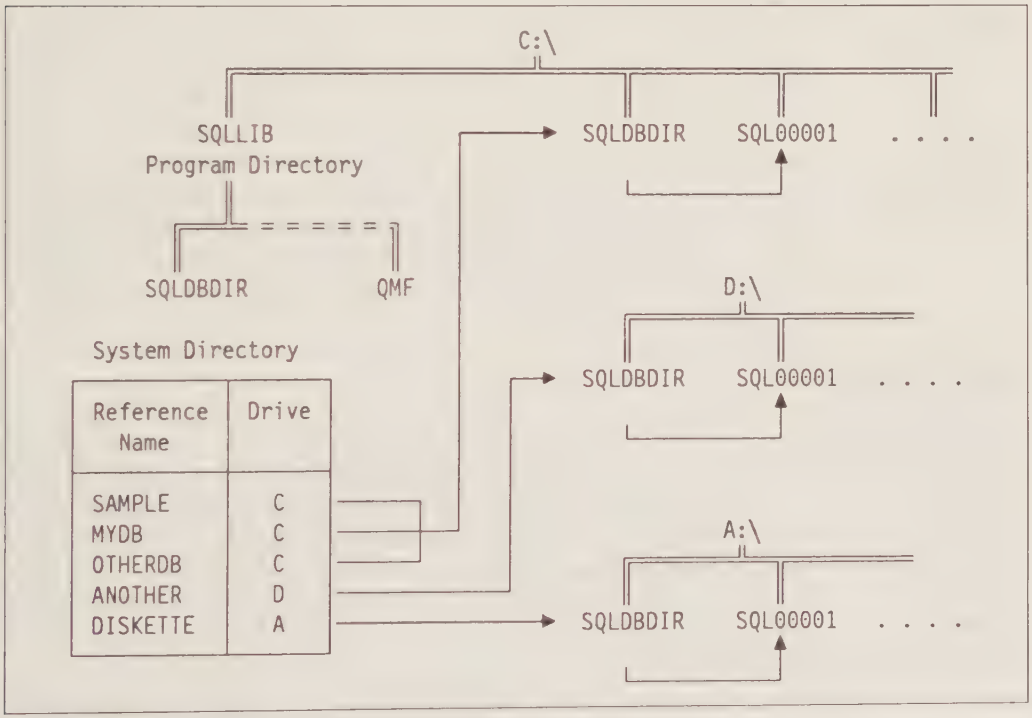



**Figure 6-2** The Database Manager System Directory

The System Directory contains a list of all the databases you can access from your PC. In order to locate the correct OS/2 directory containing the database, Database Manager scans the System Directory to determine the OS/2 drive for the database. The Database Manager then scans the Volume Directory mounted on that drive. The Volume Directory contains the correct OS/2 directory name for the requested database.

Databases created on your PC are automatically added to the System Directory, so the entire process of locating a database is hidden. If you create a database on a removable volume, such as the DISKETTE database in Figure 6-2, the System Directory entry points to the drive on which the volume was mounted. An error is returned if you mount this database on the incorrect drive. (In other words, if you create a database on the A drive then move the diskette to the B drive, Database Manager will not be able to find the database and will return an error.)

To access that database on the B drive, you must create a new entry in the System Directory that points to the B drive. Creating a new entry in the System Directory is called *cataloging* a database. Cataloging is most useful for accessing a database that was not created on your PC but was given to you on diskette. Database catalog operations are covered in "Changing the Database Directory" in Chapter 12.

One further important aspect of the System Directory is that each database has two names.

**original name** This name must be unique with respect to the volume on which the database is allocated. The disk drive value determines the volume where the database is created. The original name never changes as long as the database exists.

**reference name** This name must be unique with respect to all the databases known to exist in the System Directory on this PC. This name can be changed by the catalog and uncatalog function. There can be more than one reference name for the same database. A reference name is what a program (or Query Manager) uses to access the database; this reference name is sometimes called the *alias*.

## CONCLUSION

The Database Manager is an integral component of the OS/2 Extended Edition. As such, it relies upon the same features of OS/2 as do your applications. Your understanding of how Database Services uses features such as files and directories can assist your application's management of these same resources. The result is an application that coexists with other Database Manager applications.

This part of the book has covered several topics that deal with design of an application program which uses Database Services. The next part of the book provides details on writing an application program.

# PART THREE

# Writing Database Programs in C

In Part 3 of this book you will build C language programs that use Database Services; we assume that you already have a good understanding of the C language. Writing your own database programs allows you to exploit the Presentation Manager for user interaction and OS/2 multitasking support. In fact, you can combine all of the programming interfaces of the Extended Edition (database, communications, and Presentation Manager) in order to build a sophisticated, distributed application.

You will probably want to build your own programs if you have special data entry or output requirements. You can build an application that uses special hardware for data collection (a bar code scanner, for example) and provides an application-specific interface such as a point-of-sale terminal. Your application might be a program that provides a user interface tailored for a specific task.

Your database programs can also build in administrative tasks like regenerating statistics or reorganizing tables. Other more general interfaces such as the Query Manager leave these tasks as user responsibilities. Applications that hide these tasks are well suited for the personal computer market where the end user is unlikely to have database administration skills.

Three major topics are covered in this part of the book. Chapters 7-10 give you the fundamental knowledge necessary to build database programs that use the Database Services function calls and SQL statements. These chapters describe the model for database access, including the fundamental statements used in every database program. Details are provided on using both static and dynamic SQL statements within your C language program. Chapter 10 details database programming considerations that show how working with a database system can affect the design of your program.

The second major topic of this part is covered in Chapter 11, "Precompiling, Compiling, and Linking." This chapter discusses the mechanics of converting your database program into a running application.

The final topic of this part is covered in Chapter 12, "Utility and Environment Functions." These are some of the less frequently used Database Services functions. Chapter 12 makes recommendations regarding where those functions are better suited for execution from the Query Manager rather than within your database program.

Where possible, programming functions are explained through the use of sample programs. However, we do not list the syntax for every function. The *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference* provides this comprehensive list. This IBM guide is an invaluable reference, for it also contains a complete listing of error messages (and corrective actions) for SQL statements, function calls, and application support utilities.

The sample programs have all been written in the IBM C/2 language, version 1.1, and can be used as a programmer's toolkit for creating and manipulating databases. When complete, these programs can help you create your own database application.

# CHAPTER SEVEN

# Database Program Fundamentals

## OVERVIEW

In this chapter you will learn how to implement features common to all Database Services programs. We discuss the database program model and show you how to structure your program. This section describes how to use the Database Services function calls to connect and disconnect from your database. Additionally, we present code to move data to and from your database with SQL. The chapter concludes with a discussion on handling errors returned by Database Services. Following chapters detail the actual coding of SQL statements.

Chapters in this part of the book show you the use of Database Services functions. The C language function names are used in the text and examples describing each function. Before beginning the description, let's take a look at the function-naming conventions.

Each function name is a maximum of eight characters long. IBM has packed so much information into those eight characters that it sometimes becomes difficult to discern the function from its name. A brief explanation of the naming scheme will help you remember these names: The first three characters are always **sql**, the fourth character indicates the functional area, and the last four characters describe the function. Table 7-1 shows an example function call from each functional area, highlighting the identifying fourth character. The names of all the Database Services functions are listed in Appendix A.

## DATABASE PROGRAM MODEL

When working with data in a database (tables, indexes, views), remember this simple model for your program:

1. **Start Database Services.**

   Your application can start Database Services in one of two ways. The best way to start Database Services is to include the STARTDBM command in a command file prior to starting your application. Placing the command in STARTUP.CMD ensures that Database Services is always started at system power up or reset. STARTDBM accomplishes the desired function so that you need not program the function. You can check the exit code with the OS/2 batch command IF ERRORLEVEL. You can also use the Start Database Manager function call **sqlestar** to start Database Services from within your

| Area | 1–3 | 4 | 5–8 | Description |
|---|---|---|---|---|
| Environment | sql | **e** | cred | Create Database |
| Utilities | sql | **u** | imp | Import Table |
| Configuration | sql | **f** | xdbc | Return Database Configuration |
| Application | sql | **a** | bind | Create Access Plan |

**Table 7-1** Function Naming Conventions

program. Attempting to start Database Services after it is already started causes no harm. This function call is documented in the *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference*.

2. **Connect to a database.**

   The Start Using Database function call **sqlestrd** establishes a connection to a specific database.

3. **Do your work.**

   Once connected, your program may execute SQL statements to create and manipulate data in the connected database.

4. **Stop using a database.**

   The Stop Using Database function call **sqlestop** disconnects your application from the active database.

Every time your application makes a request of Database Services, whether through a function call or an SQL statement, Database Services returns information on the completion of that function. The "Error Handling" section in this chapter discusses how to handle this return information in your application.

## Connect to a Database

Before you can do any work with data within a database, you must make a connection to that database. The function call to make this connection is named Start Using Database. The following program fragment contains an example of the **sqlestrd** function and is typical of the type of code you should write to connect to your database.

```
sqlestrd(argv[1],password,SQL_USE_SHR,&sqlca);
if (sqlca.sqlcode == SQLE_RC_DB_RESTART)
   {
    printf("Database Restart in progress. Please wait...\n");

    step = "Restart Database";
    sqlerest(argv[1],password,&sqlca);
   }
```

The first parameter, argv[1], is a pointer to an ASCIIZ string containing the name of the database to connect. In this case the database was passed as the first argument when the program was invoked.

The second parameter, password, is a pointer to an ASCIIZ string containing the database password. The string is ignored if no password was specified when this database was created. You can avoid asking the user to supply a null password if you attempt the database connection first. If no password is required, you are ready to begin work. If a password is required, a return code indicates the failed connection. You can then prompt for the correct password and retry the connection.

The third parameter, SQL_USE_SHR, is a Database Manager constant indicating that the program allows others to share the database. In general, you should always request shared access to allow other applications concurrent use of the data. Implicit locking and transaction management provided by Database Services maintain the consistency of the changes you make. Exclusive use should be used when other concurrent applications would be disruptive. For example, the Database Services reorganize utility requests exclusive use of the database to guarantee that the reorganize can complete. This means you cannot be connected to a database containing a table being reorganized. See Chapter 12, "Utility and Environment Functions," for more information on reorganization.

The last parameter, &sqlca, is a pointer to a structure for return code information. This structure is described in "The SQL Communications Area" in this chapter.

The sample code above checks for a specific return code indicating that the database needs to be restarted. This condition can occur if the database was left in an inconsistent state, which can arise from an abnormal termination while a program was connected to that database. The database can be returned to a consistent state through the use of the Restart Database function call, **sqlerest**, which accepts parameters similar to Start Using Database.

Your application can proceed if you receive a successful return code.

## Using SQL Statements

You create and manipulate data within your database by embedding SQL statements within your C language program. Since the C compiler is unable to process these statements, the Database Manager provides a utility to convert SQL statements to C. This utility is called the *precompiler*. Precompilation is discussed further in "Placing SQL Statements in Your Program," in this chapter, and in "Precompile and Bind" in Chapter 11. The following sections describe how to place SQL statements into your C program and move data between these SQL statements and your program.

### *Placing SQL Statements in Your Program*

You must delimit SQL statements within a C program so that they can be identified by the precompiler. Precede each SQL statement with an "EXEC SQL" and terminate it with a semi-colon (";"). The delimiters are highlighted in the following example:

```
EXEC SQL INCLUDE SQLCA;
```

You can split longer statements across lines within your program file. You can also include comments within your SQL statements as shown in this example:

```
EXEC SQL CREATE staff (ID     SMALLINT NOT NULL, -- employee id
                       NAME   VARCHAR(9),        -- name
                       DEPT   SMALLINT,          -- department number
                       JOB    CHAR(5),           -- job title
                       YEARS  SMALLINT,          -- years with company
                       SALARY DECIMAL(7,2),      -- current salary
                       COMM   DECIMAL(7,2))      -- commission;
```

Here is a useful debugging tip: Be sure that all of your C language strings and comments are properly delimited. The Database Manager precompiler uses these to know whether the embedded SQL statement should be processed. (In other words, the precompiler does not process a SQL statement that it thinks is contained within a C language string or comment.) Mismatched C language delimiters might make the precompiler think your SQL statement is in a comment. These errors are detected when running the C precompiler.

### *Moving Data To and From the Database*

One of SQL's functions is to retrieve data. Within your C program, this means getting data from the database into C variables that you can further manipulate with your program. You can retrieve data simply by placing a C variable in an SQL statement. C variables used in this manner are called *host variables*. Host variables can also pass data to SQL. For example, you may want to use a host variable on an INSERT statement or as part of the WHERE clause of a SELECT statement.

You must declare these variables in a special place within your program so that the precompiler can recognize them. The precompiler needs to know the declared data type of the host variable in order to perform appropriate data conversions when moving data to or from the database table. Additionally, the precompiler uses this information to perform error checks on your SQL statements, verifying that appropriate host variable data types are used for table column values.

This section of your program is called the SQL DECLARE SECTION. The following example shows the declarations for the STAFF table of the IBM SAMPLE database. Notice that the declarations are between BEGIN DECLARE SECTION and END DECLARE SECTION SQL statements.

```
EXEC SQL BEGIN DECLARE SECTION;

      short          id;
      unsigned char  name[10];
      short          dept;
      unsigned char  job[6];
      short          nyears;
      double         salary;
      double         comm;

EXEC SQL END DECLARE SECTION;
```

Host variables can be any of the C storage classes: extrn, static, or automatic. Therefore, you can declare host variables in main(), outside main(), or as local variables within a subroutine. Declaring your host variables in main() allows you to gather your application variables within a single source module, yet reference them with SQL statements from all other source modules (where they are declared extrn). Support for automatic storage class variables allows you to create a library of C routines that process SQL statements with host variables that have been passed in the routine's parameter list.

You can have multiple declaration sections within a single source file. You may find this useful when building a library of SQL routines where each routine uses local variables with its SQL statements (accepts returns host variables on the stack). The following code depicts this situation.

```
/*****************************************************************/
int insert_salary_&_commission(id, salary, comm)

EXEC SQL BEGIN DECLARE SECTION;

     short id;
     double salary;
     double comm;

EXEC SQL END DECLARE SECTION;

     /* program body */
/*****************************************************************/
int insert_years(id, nyears)

EXEC SQL BEGIN DECLARE SECTION;

     short id;
     short nyears;

EXEC SQL END DECLARE SECTION;

     /* program body */
```

The SQL column type and the C variable type of the host variable used to hold the contents of that column must match. Table 7-2 shows the appropriate C data types for each column data type. Database Services performs appropriate conversions should you use a data type other than those shown in the figure. For example, Database Services truncates the decimal portion of a FLOAT column when FETCHed into a C integer.

If a column is nullable (that is, not created as NOT NULL), an extra host variable is required to hold the column's value. This extra host variable is called the *null indicator*. A null value requires an extra variable since a null value cannot be indicated by a 0 or blank value in the column.

The following SQL DECLARE SECTION is taken from the sample program DBSTRUCT.SQC. It shows the declaration of host variables for several nullable columns. The sample program adopts a convention of naming null indicators the same as their corresponding column host variables, only with the "_nid" suffix. This convention makes null indicator names easy to remember.

| For This SQL Data Type . . | Use This C Data Type | Description |
|---|---|---|
| CHAR(1) | char; | Single character. |
| CHAR(n)<br>1 < = n < = 254 | char[n + 1];<br>char[n]; | null terminated<br>NOT null terminated<br>Fixed length character string. |
| DATE | char[10] | SQL date column. |
| DECIMAL(p,s) | double; | Packed decimal. C does not support decimal data types. Use floating point variables to hold data from these columns. Precision may be lost in the conversion from packed decimal to double. If you need the precision of the DECIMAL data type use dynamic SQL to get the packed decimal value and then convert. |
| FLOAT | double; | Double precision (IEEE 8 byte) floating point. |
| INTEGER | long;<br>long int; | 32 bit signed integer. |
| Null Indicator | short; | 16 bit null indicator |
| SMALLINT | short;<br>short int; | 16 bit signed integer |
| TIME | char[8] | SQL time column. |
| TIMESTAMP | char[26] | SQL timestamp column. |
| VARCHAR(n)<br>1 < = n < = 4000 | char[n + 1];<br>char[n]; | null terminated<br>NOT null terminated<br>Varying character string |
| LONG VARCHAR(n)<br>1 < = n < = 32700 | char[n + 1];<br>char[n]; | null terminated<br>NOT null terminated<br>Long varying character string. |

**Table 7-2** SQL to C Data Type Mapping

```
EXEC SQL BEGIN DECLARE SECTION;

unsigned char tcreator[9];    /* SYSTABLES.CREATOR */
unsigned char tname[19];      /* SYSTABLES.NAME */
unsigned char tcomment[255]; /* SYSTABLES.REMARKS */
short tcomment_nid;
unsigned char cname[19]       /* SYSCOLUMNS.NAME */
unsigned char ccomment[255]; /* SYSCOLUMNS.REMARKS */
short ccomment_nid;

EXEC SQL END DECLARE SECTION;
```

Database Manager places some restrictions on the declarations supported in the SQL declaration section. These restrictions are listed below.

1. The IBM precompiler does not support user defined data types (typedef). You must declare all host variables as base C language data types. Table 7-2 lists C data types that

can be used to hold SQL data types used with Static SQL. Chapter 9, "Using Dynamic SQL," contains information on using host variables with Dynamic SQL.

2. The IBM precompiler does not support structures as host variables. See the use of pointers below for a way to place SQL data into structures.

3. Do not use macros and defined symbols since the precompiler does not resolve these references. This means that all numbers, such as array sizes, must be hard-coded within the declaration section. See "Precompile and Bind" in Chapter 11 for a way to avoid this restriction.

4. You may use pointers in SQL statements. Define the pointer within the SQL declaration section. The data *pointed to* can be anywhere within your program. Pointers as host variables are useful for placing SQL data into C structures and arrays (of any data type). For character data, the declaration of the pointer within the SQL declaration section must contain the length of the data being pointed to. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
     char (*name)[10];
EXEC SQL END DECLARE SECTION;
```

   allows you to use the pointer name within your SQL statements. You may use pointers in SQL statements to place data in structures (outside the SQL declaration section). You may then use these structures to reference your data outside of the SQL statements (you must still use the pointers declared in the SQL declaration section within your SQL statements).

5. All host variables are global to the precompiler. Thus, even if a host variable in a declare section is local to a procedure, it can only appear in ONE SQL declaration section per source module.

### Stop Using a Database

After your work with the database is complete, you must disconnect (Stop Using) the currently connected database. Use the **sqlestop** function call to do this. At this time Database Services performs an implicit COMMIT on behalf of your transaction. The only parameter accepted on **sqlestop** is a pointer to the return code structure.

Database Services automatically stops using the current database when your application terminates and returns to OS/2. However, you should always use the **sqlestop** function call in order to process any errors that might be encountered during the stop function.

## ERROR HANDLING

Although we did not mention it as one of the four steps in the database program model, error handling is an important part of every database program. You will find that the use of SQL in your program can dramatically increase the number of error conditions your

program must handle. In the following sections you will learn how Database Services returns error information to your program and how you can use the functions to make programming responses an easier job.

### The SQL Communications Area

A fixed-size data structure called the the SQL Communications Area (SQLCA) returns error code, warning, and diagnostic information to your program. The SQLCA is used to return information from both function calls and SQL statements. IBM provides a structure defining the SQLCA in the include file **sqlca.h**. The following code shows the structure of the SQLCA.

```
struct sqlca
{
   unsigned char sqlcaid[8];   /* Eyecatcher = 'SQLCA ' */
   long          sqlcabc;      /* SQLCA size in bytes = 136 */
   long          sqlcode;      /* SQL return code */
   short         sqlerrml;     /* Length for SQLERRMC */
   unsigned char sqlerrmc[70]; /* Error message tokens */
   unsigned char sqlerrp[8];   /* Diagnostic information */
   long          sqlerrd[6];   /* Diagnostic information */
   unsigned char sqlwarn[8];   /* Warning flags */
   unsigned char sqlext[8];    /* Reserved */
};
```

Your program must include an SQLCA for Database Services to use. The *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference* contains a good description of all the fields in the SQLCA.

You can use the following statements to include an SQLCA in your program:

```
#include <sqlca.h>
struct sqlca sqlca;          /* my SQL Communications area  */
```

The name you use for your copy of the SQLCA is important. When using SQL statements, Database Services expects your SQLCA to be named **sqlca**, the name used in the statements generated by the precompiler. So, if you use the method above, you *must* name your copy of the SQLCA **sqlca**.

There is an easier way to include an SQLCA in your application. An SQLCA is declared with the appropriate name if you place the following statement in your program:

```
EXEC SQL INCLUDE SQLCA;
```

This SQL statement directs the precompiler to generate the appropriate C language statements to declare an SQLCA with the name **sqlca**. If you use this statement, you need not use the supplied include file (**sqlca.h**) or reserve storage for an SQLCA because these statements are generated by the precompiler.

When you use a Database Services function call, you must tell Database Services the location of your SQLCA. You do this by passing a pointer to the SQLCA on each of the function calls.

You can save yourself some work by always using the INCLUDE SQLCA statement to generate your copy of the SQLCA. By doing so, you can be sure that the variable **sqlca** is always a valid SQLCA. All of your programs, whether they use SQL statements or not, must be precompiled (to expand the INCLUDE SQLCA statement), but all of your database programs now go through the same development process.

When you issue an SQL statement, return information is automatically placed into an SQLCA with the name **sqlca.** Place the INCLUDE SQLCA statement outside of your main() routine in order to allow all of your routines reference to the same SQLCA. Simple, single thread applications require only one SQLCA, so if your application comprises many modules, declare the SQLCA only once and define it with the extrn storage class in all other modules. You must declare these extrn references to the SQLCA itself since every invocation of the INCLUDE SQLCA statement generates a new copy of an SQLCA.

The next section shows how to check for errors and what to do if one occurs.

## Using the SQLCA

Let's look at an example of using one of the Database Services function calls. Below is a fragment of a C program that issues the Start Using Database routine. For now, concern yourself with only two things: the return code (rc) and the pointer to the SQLCA (**&sqlca**).

```
rc = sqlestrd(database,password,use,&sqlca);
```

Each Database Services routine returns an integer value that indicates the condition of the SQLCA, whose address was passed by **&sqlca.** A return code of zero indicates that the SQLCA is good and can be examined by your program. A non-zero return code indicates that an invalid SQLCA was passed to the function. In this case, your program should not examine the SQLCA passed on the call.

When a function call returns to your program, you should first check this return code and then check the return code in the SQLCA. The next two sections show examples of how to check return codes in the SQLCA.

### *The WHENEVER Statement*

You should also provide a check of the sqlca.sqlcode after every SQL statement in your program. This is very inconvenient if you have more than a few SQL statements. Since each statement is processed by the precompiler, it would be helpful for the precompiler to generate some code to check the sqlcode. The Database Manager provides the WHENEVER statement for just this purpose.

There are three versions of the WHENEVER statement; each one is for a specific set of return codes.

```
EXEC SQL WHENEVER SQLERROR GO TO label;     OR
EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL WHENEVER SQLWARNING GO TO label;   OR
EXEC SQL WHENEVER SQLWARNING CONTINUE;

EXEC SQL WHENEVER NOT FOUND GO TO label;    OR
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

Each statement can take one of two forms:

1. CONTINUE tells the precompiler to *not* generate code to check for the specified condition.
2. GO TO label tells the precompiler to generate code that transfers program control to the specified statement. GO TO may also be written GOTO.

Unfortunately, the transfer of control is via a GO TO, rather than a call return interface. Therefore, you can not identify where in your application the WHENEVER was invoked from, nor is there a way of returning execution control back to that point in the application.

For these two reasons you should consider writing a routine that does common error checking of the SQL return code. This routine can be link edited with all your database programs. You would then call this routine after every SQL statement and Database Services function call you execute. Be sure to omit the SQL WHENEVER statement so that the precompiler does not generate any GO TOs that would conflict with your call to the common error routine.

There are several reasons a common routine is beneficial and they are listed below.

1. **Code Reuse** Recoding the same statements repeatedly is inefficient. You can write a common error checking routine once and reduce your development effort.
2. **Reduced Testing** If your code works, you will want to reduce testing the same error handling in different parts of your application.
3. **Improved Diagnostics** When errors occur, you can get helpful debugging and error determination information in a common and consistent format.
4. **Migration and Portability** Since SQL error conditions may change with new product releases, it is easier to change one error routine than every SQL statement's error routine. You may also want to run the application against a different SQL product. Even if this is an IBM SQL product, they all have some product-specific return codes.

The following example shows how the coding might be done for this routine.

```
#define MOD_ID 36       /* logical module ID */

EXEC SQL INSERT INTO TAB1 VALUES .......;
   result = SQL_Error_Routine(sqlca,MOD_ID,21 . . . );
   if (result != 0)
```

Your error routine might have some of the following input parameters:

| PARAMETER | FUNCTION |
|---|---|
| **sqlca** | The sqlca used by the SQL statement. The errror code returned is in this data area. |
| **COMP-ID** | This ID identifies the application or application component of the calling routine. It can be used in an error message or application log record to more quickly determine the location in the application where the failure has occurred (for example, it might be a unique source module ID). |
| **STMT-ID** | This ID identifies and locates the SQL statement that failed. |
| **ecodes** | A pointer to a vector of error codes that the application wants to handle. This provides a capability for the application routine to override the error routine in handling specific SQL errors.<br>Any application that expects to share data with other applications can be victim of the dreaded DEADLOCK condition. There is nothing you can do in your application to prevent deadlocks from happening; even the most well behaved transaction can be involved in a deadlock. Thus, you should choose some strategy for handling this situation. In some cases, the simplest solution will be to restart your application transaction. Other applications may be more complex and might let the common error routine post an error message and terminate the application with a nonzero OS/2 exit code.<br>A more common situation is to get an error inserting a row that contains a duplicate key for a unique index. Let Database Services tell you of the error instead of doing the additional programming to check the condition before attempting the INSERT. |
| **num_ecodes** | The number of error codes in the ecodes vector. |
| **environment** | You can provide a parameter to indicate the environment in which the application is running. This is one technique for dealing with different SQL products and environments. |
| **err_category** | Another technique for handling error codes is to have the common routine map the SQL error code into a reduced set of error codes. You can receive any of over 1,000 error codes. They can be classified as follows:<br>1. **Programming errors** These comprise the largest group of error codes. A common routine can put up an error message and abort the application.<br>2. **Environmental errors** These errors are out of the application's control. This includes errors concerning such things as Database Services buffer or heap space that may be caused by either too many OS/2 or |

Database Manager applications. Hardware errors should also be considered in this category. A common routine can put up an error message and abort the application.

3. **Recoverable errors** By definition these are errors that some applications can recover from with or without user assistance.

The error routine could return a result indicating whether the error was recoverable and let the application decide whether user assistance is required. For example, the log full error condition (SQL error code -964) is recoverable via one of several options: retry, COMMIT, or ROLLBACK. Your application can ask the end user which option is desired since the recovery steps are quite different for each.

## Error Messages

After determining that an error or warning has occurred (sqlca.sqlcode is non-zero), your program should process the error. At a minimum, your program should notify the user of the error so that the condition can be corrected. Your program could test sqlca.sqlcode against all the possible errors and display a message for each, but writing a routine to do this would be a large task considering the number of possible error return codes. Instead, Database Services provides a special routine to retrieve error message text for your program. This function is named **sqlaintp.**

The **errmsg()** routine in the following code uses the **sqlaintp** routine to generate an ASCII error message corresponding to the error code in the SQLCA. This routine is used by the sample Database Services function call routines described in Chapter 12, "Utility and Environment Functions."

```
/*****************************************************************/
/*                          E R R M S G                          */
/*****************************************************************/
void errmsg()
{
#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];

  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);
                                    /* get error msg text   */
  printf("%s\n",buffer);            /* print msg text       */
  return;
} /* end errmsg */
```

In practice, **sqlaintp** is very easy to use and understand. Provided with a pointer to the SQLCA containing the error, **sqlaintp** returns ASCII text for the error message. The text is placed into a buffer that you supply, and the text is formatted into lines of your specified length.

***Note:*** The **sqlaintp** routine returns an integer error code. It does not modify the SQLCA, so you should check only the function return code for information on the success of the error message retrieval; do not recheck the sqlca.sqlcode. The *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference* lists the return codes generated by **sqlaintp.**

The **sqlaintp** routine does much more than simply look up the text from a file. Other error information from the SQLCA that is specific to your error is incorporated into the message text. For example, if the Start Using Database function call were executed with a file system drive letter that did not exist on your system, say "X:", the message returned would be:

```
-1002 SQL1002N "X" is not a valid drive.
```

The real power of this routine becomes visible when SQL statements return processing errors. One SQL error, invalid syntax, returns an error message identifying the portion of the statement in error. Additionally, a partial list of valid statement tokens is provided, as follows:

```
-104 SQL0104N An unexpected token "xxxxxxx" was found following
      "xxxxxxxx". Expected tokens may include "xxxxxxxx".
```

The variable message text is returned in the sqlerrmc field of the SQLCA. The **sqlaintp** routine uses these variable tokens to build the message returned to your program.

Besides relieving you of the need to provide error messages for all the possible errors returned by Database Services, **sqlaintp** saves you from having to translate message text should you wish to provide language specific versions of your program. IBM translates the message file.

This error message routine is handy for debugging purposes, but you may find your use of it limited for the following reasons. First, you cannot customize the messages returned by Database Services. These messages can be quite technical and not particularly meaningful to an end user of your application. Second, the Database Services message facility returns only a formatted buffer of data. Your program must still display the message text, unlike the Presentation Manager message facility, which also displays returned message text.

## Handling Ctrl + Break

Your database application must handle Ctrl + Break signals generated when the user presses Ctrl + Break or Ctrl + C. OS/2 notifies your application that Ctrl + Break has been pressed only if you have installed a Ctrl + Break signal routine. Database Services provides two function calls to assist your handling of the Ctrl + Break signal.

Processing Ctrl + Break is important. Suppose your application does not install a Ctrl + Break signal handler. If the user presses Ctrl + Break, OS/2 terminates your

program and your application does not have the chance to complete any transactions that were being processed. The Database Manager then marks the database as invalid and terminates all processes connected to that database. The database must be recovered with a Database Restart.

Database Services provides a function call that can be used to install a default Ctrl + Break signal handler, **sqleisig.** Simply issue this call before any other Database Manager operation and your database's consistency will be protected from Ctrl + Break. The default signal handler supplied by IBM catches Ctrl + Break or Ctrl + C and issues a special function call to Database Services. This call is the Interrupt routine, **sqlaintr.** This routine interrupts and ends gracefully any Database Manager operation currently in progress.

```
sqleisig(&sqlca);    /* Install Default Signal Handler */
```

The default signal handler is adequate for very simple, single task applications. If you build a multithread or process application, you should process Ctrl + Break yourself. Other situations may require switching between several Ctrl + Break handlers depending on the function being executed by your application.

When building your own Ctrl + Break handler, be sure to use the interrupt routine, **sqlaintr**, as part of the handler. A typical Ctrl + Break handler examines some state information about the executing process in order to determine if the Ctrl + Break should be honored. If terminating the process is allowed, the handler "cleans up" and then returns to OS/2. You should call the interrupt routine as part of this clean up step.

The interrupt routine notifies Database Services that a termination has been requested. This interrupt allows Database Services to examine what, if any, database operation is in progress and take appropriate action to terminate that operation. Some database operations such as COMMIT or ROLLBACK cannot be terminated and are allowed to complete since they are necessary to guarantee the consistency of the data. The *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference* contains a complete list of actions taken when specific database operations are in progress.

## SUPPLIED INCLUDE FILES

When using SQL and the Database Services functions in your program, you must provide definitions of each of the routines to be called. IBM supplies several C language files that you must include in your program source in order to provide these definitions. Each of the sample programs has a section of #include statements that reference these C language include files. Here is an example from CREATEDB.SQC:

```
#include <stdio.h>              /* Standard C I/O functions     */
#include <string.h>             /* C string manipulation fns    */

#include <sqlenv.h>             /* SQL Environment include      */
#include <sql.h>                /* SQL constants & err msg fn   */
```

The first two statements are required for the standard C functions used in each program. The last two are required in order to use the Environment routines (sqlenv.h contains the definition of sqlecred) and the error message retrieval routine (sql.h contains the definition of sqlaintp). If your program requires one of the Utility or Configuration routines, it must include the following statement:

```
#include <sqlutil.h>          /* SQL Utility/Config include */
```

The following include files are available for your use in C programs. The Database Manager installation process places these files in the C:\SQLLIB directory.

| FILE | FUNCTION |
|---|---|
| **sql.h** | Defines system constants such as the maximum number of key parts allowed in an index and SQL data type size limitations. This file also includes the function prototypes for the Application Support routines. |
| **sqlca.h** | Defines the SQLCA structure. |
| **sqlcodes.h** | Defines the possible values for the sqlcode field of the SQLCA that may be returned after processing an SQL statement. |
| **sqlda.h** | Defines the SQLDA structure. |
| **sqlenv.h** | Contains the function prototypes for the Environment routines. Also contains constant, return code, and structure definitions that may be used in conjunction with the Environment routines |
| **sqlutil.h** | Contains the function prototypes for the Utility and Configuration routines. Also contains constant, return code and structure definitions that may be used in conjunction with these routines. |

## BUILDING A SAMPLE DATABASE PROGRAM

This section presents a simple database application that summarizes the database program fundamentals discussed in this chapter. This section also provides additional detail on the use of SQL for retrieving data from a database. TUTORIAL.SQC, the sample program below, has specific elements labeled to correspond with the explanations, which follow the listing.

After the elements are described, we guide you through the steps necessary to turn this program into a running application. If you follow all of these steps, you will have a running application that lists all the tables in the database SAMPLE.

```
#include <stdio.h>
#include <sql.h>              /* SQL constants                    */
#include <sqlenv.h>           /* SQL Environment routines         */
#include <sqlcodes.h>         /* SQL return code definitions      */

EXEC SQL INCLUDE sqlca;       /* include SQLCA structure          */

EXEC SQL BEGIN DECLARE SECTION;
  unsigned char table_name[19];
EXEC SQL END DECLARE SECTION;

int main()
{
  sqlestrd("sample","",SQL_USE_SHR,&sqlca); /* use SAMPLE       */

if (sqlca.sqlcode != SQL_RC_OK) {          /* if SQL error     */
   printf("Error opening SAMPLE - %ld.\n",sqlca.sqlcode);
   return(2);
}
EXEC SQL WHENEVER SQLERROR GOTO sql_error;

EXEC SQL DECLARE current_row CURSOR FOR
         SELECT NAME FROM sysibm.systables ORDER BY NAME;

EXEC SQL OPEN current_row;
EXEC SQL FETCH current_row INTO :table_name; /* get first rw */
while (sqlca.sqlcode == SQL_RC_OK) { /* while rows exist     */
       printf("%s\n",table_name);
       EXEC SQL FETCH current_row INTO :table_name;
       }
EXEC SQL CLOSE current_row;       /* close the cursor         */

goto end;
sql_error:
  printf("SQL Error encountered - %ld.\n",sqlca.sqlcode);

end:
  sqlestpd(&sqlca);               /* stop using SAMPLE        */
  return(0);
}
```

The database program elements are listed below.

**1 Include files.**

The sample program starts by including the files that contain definitions of SQL constants and return codes along with the definitions of the Database Services function calls necessary to work with your database. In general, files supplied by IBM for the Database Manager begin with SQL.

**2 INCLUDE SQLCA.**

This embedded SQL statement is delimited with EXEC SQL and a semicolon (;). After precompilation, this SQL statement generates a C structure for the SQL Communications Area. This structure is named sqlca.

**3 SQL Declaration Section.**

In order to get data into or from your database, you must use C language variables in your SQL statements. These C variables are called host variables. All host variables must be declared inside the SQL declaration section, delimited by the BEGIN and END DECLARATION statements. Host variables are placed here so that the SQL precompiler can find and process them. In this sample, the returned table name is placed into a C character variable, table_name.

**4 Start Using a Database.**

Before you can begin working with the data in a database, you must tell Database Services the name of the database that contains your data. This program uses one of the Database Services function calls and connects the IBM SAMPLE database.

**5 Error Handling.**

Your program should check for errors after each call to Database Services. This program checks the SQLCA for an error immediately after the call to Start Using a data base. The SQL WHENEVER statement tells the precompiler to generate code to check for errors after each SQL statement. In this program, the precompiler is told to generate code that transfers control to **sql_error** when an error is detected.

**6 Declare Cursor.**

Working with SQL allows you to specify the data you want to work with, not how to get it. You use the DECLARE CURSOR statement to tell Database Services what you want. A SQL SELECT statement is assigned to a cursor to describe the set of rows to be retrieved. Since you do not want all rows to be returned at once, you use the cursor to indicate which row is currently available.

In this program, the SQL SELECT statement retrieves the Name column from the SYSIBM.SYSTABLES table (this is the system catalog table that lists all of the tables in a particular database). This is also the time to tell Database Services that you want the table names returned in alphabetical order.

**7 Retrieve Data Loop.**

The program must now retrieve each row of data from the database. Since you already told Database Services the data you want, simply retrieve every row described by the cursor, one at a time. This WHILE loop does just that. After OPENing the cursor and FETCHing the first row, a loop is entered that prints the result of the FETCH (the host variable table_name). This action is repeated as long as a successful return code (SQL_RC_OK) is encountered. The loop terminates when the end of data condition (or any other condition) is returned.

**8 Error Messages.**

In response to the SQL WHENEVER statement above, control transfers to this point in the program if an error is encountered while processing the rows of data. This program prints out the error code and then terminates the program.

**9 Stop Using the Database.**

Before leaving the program, be sure to Stop Using the database. Another Database Services function call is used here to disconnect from the IBM SAMPLE database.

## How to Precompile, Compile, and Link the Example

Follow these steps to create and run the sample program.

1. **Write your source program.**

   Using an ASCII editor (such as the OS/2 System Editor, "E"), type the statements from the sample program. The file name must have an extension of .SQC (a C language SQL program). Let's assume you choose TUTORIAL.SQC and you create this file in a directory named C:\TESTPGM.

2. **Start Database Services.**

   Use the STARTDBM command to start Database Services. You can execute this command from your current directory since the Database Manager program directory was added to the OS/2 path when you installed Database Services.

   ```
   [C:\TESTPGM]startdbm
   ```

3. **Create the SAMPLE database.**

   If you have not already done so, type the following command to create the IBM SAMPLE database:

   ```
   [C:\TESTPGM]sqlsampl
   ```

4. **Precompile your source program.**

   Precompile TUTORIAL.SQC with the following statement:

   ```
   [C:\TESTPGM]sqlprep tutorial.sqc sample
   ```

5. **Compile the precompiler output.**

   Type the following command to run the IBM C/2 compiler against the C source file output by the precompiler. Be sure to type each option in upper- or lowercase as shown; the case of each option is significant.

   ```
   [C:\TESTPGM]cc tutorial.c /AL /W2;
   ```

6. **Link your object modules.**

   The compiler generates the object module, TUTORIAL.OBJ. You can now link this program into a running program with the OS/2 link command. Enter these responses at the appropriate link command prompts:

```
[C:\TESTPGM]link

IBM Linker/2 Version 1.10
(C) Copyright IBM Corporation 1984, 1988
(C) Copyright Microsoft Corp 1983, 1988. All rights reserved.

Object Modules [.OBJ]: TUTORIAL
Run File [TUTORIAL.EXE]:
List File [NULL.MAP]:
Libraries [.LIB]: LLIBCP+DOSCALLS+SQL_STAT+SQL_DYN
Definition File [NUL.DEF]: /NOD
```

7. **Execute your application.**

   You can now run TUTORIAL.EXE as follows:

```
[C:\TESTPGM]tutorial
```

   A list of table names in the IBM SAMPLE database is displayed on your screen.

## CONCLUSION

This chapter has covered the basic elements of a database program. Every program must be able to start Database Services, connect to a database, do some work, and then disconnect that database. Error handling was demonstrated using the SQL WHENEVER statement as well as through a common error handling routine. These are all key elements that are required when working with data in Database Manager databases.

Using the sample program in this chapter you may have written some basic SQL statements that retrieve data. These and other SQL statements are the topic of the next chapter.

# CHAPTER EIGHT

# Using SQL Statements

## OVERVIEW

In Chapter 2, "What is a Database Application?", you learned about the four elements that comprise a database application—a high-level programming language, Database Services function calls, the Structured Query Language, and the Query Manager. This chapter takes a detailed look at combining these elements into practical examples. As you know, SQL statements are embedded within the IBM C/2 language to create and manipulate data within databases; these are the statements you use to actually do your work. SQL statements that are unique to the programming environment are also discussed. The sample programs discussed in this chapter also use many of the Database Services function calls discussed in Chapter 7, "Database Program Fundamentals."

In this chapter you will go through step-by-step examples and learn to:

- Create tables and insert data
- Retrieve data from tables
- Update data in tables

The statements discussed here are from sample programs listed in Appendix B, "Sample Programs." If you are anxious to begin work, follow the instructions at the beginning of that appendix to enter and run the sample programs.

## DEFINING DATA OBJECTS

Database applications can be divided into two categories—those that define their own data structures and those that operate on any database structure (utilities). Those that define their own structure must use the SQL Data Definition Language (DDL) to create the database objects. Typically, these statements are part of your application installation.

The following sections step you through a set of typical SQL DDL statements that might be used in an application installation program. The statements shown are used to create the necessary tables and indexes and then fill the tables with data.

## Creating Tables

Use the SQL CREATE TABLE statement to create the structure for the tables in your database. The following CREATE TABLE statement is used in the sample program INSTALL.SQC. This statement replicates the structure of the STAFF table in the IBM sample database SAMPLE.

```
EXEC SQL CREATE TABLE staff (ID      SMALLINT NOT NULL,
                             NAME    VARCHAR(9),
                             DEPT    SMALLINT,
                             JOB     CHAR(5),
                             YEARS   SMALLINT,
                             SALARY  DECIMAL(7,2),
                             COMM    DECIMAL(7,2));
```

## Creating Indexes

You can create an index with the SQL CREATE INDEX statement, which is embedded directly into your program source. You can only create an index on an existing table. The following example creates an ascending index on the NAME column of the staff table defined above.

```
EXEC SQL CREATE INDEX ON STAFF (NAME);
```

After you create an index, future references are automatically decided by the Database Services optimizer. Later, when your application references the indexed table, you need not include an index name within your program. For this reason, indexes are usually created as part of the installation process. See "Indexes" in Chapter 3 for information about using indexes.

## Initializing Table Data

Next, initialize your table with data. Depending on your application needs, you can do this at installation (application shipped with data) or at runtime (using data supplied by the user). This second case is the more prevalent instance, but we discuss both methods.

If your application requires data to be loaded into the database as part of the installation, you can use a series of INSERT statements during that installation. The sample program, INSTALL.SQC, uses this method with statements like this:

```
EXEC SQL INSERT INTO staff
     ( ID, NAME, DEPT, JOB, YEARS, SALARY, COMM)
     VALUES ( 20, 'Pernal', 20, 'Sales', 8, 18171.25, 612.45);
```

These INSERT statements specify the columns to be filled and the values with which to fill them. At a minimum, you must specify every column that requires data (created as NOT NULL). In order to INSERT a row containing a null column you can omit that column from the INSERT statement or specify a host variable and a null indicator.

A more economic approach to loading a database at installation is to provide the data in one of the supported IMPORT/EXPORT formats and then use the import utility function call to load the table. See Chapter 12, "Utility and Environment Functions," for information on programming data import/export. The IXF format is most convenient since the table (and indexes) can be created as a result of the import. Alternately, you can have the end user use the Query Manager to do the import.

Typically your application accepts user input for insertion into a table. In this case the INSERT statement is coded with host variables containing the row data to be inserted. The data type of the host variable must be of the appropriate type for the receiving column. Table 7-2 shows the C data types that can be used for data insertion.

### *INSTALL.SQC*

The sample program INSTALL.SQC performs a sample application installation. INSTALL.SQC creates the table STAFF in the database specified on the command line. Two sample records are inserted into this new table by the **filltable()** routine, which is in FILLTB.SQC. These INSERT statements must be in a separate source module in order to accommodate the binding restrictions (see "Binding's Effect on Program Design" in Chapter 10 for more details on binding).

The SQL statements in this sample program are very straightforward. Of more interest are the support functions surrounding these SQL statements. First, INSTALL.SQC creates the database requested from the command line INSTALL.SQC then begins using the new database. Before any SQL statements can be executed, you must bind INSTALL to the new database. Use the **sqlabind** function call to perform the bind as shown here.

```
/************************************************************************/
/* Bind this program to the new database                                */
/************************************************************************/

step = "Bind Application to Database";
sqlabind("install.bnd",argv[2],password,"",NULL,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)       /* error on rebind     */
   goto exit_error;
```

Now that you have bound INSTALL to the database, create the table with the SQL CREATE TABLE statement.

```
/************************************************************************/
/* Create the sample table                                              */
/************************************************************************/

EXEC SQL CREATE TABLE staff (ID     SMALLINT NOT NULL,
                             NAME   VARCHAR(9),
                             DEPT   SMALLINT,
                             JOB    CHAR(5),
                             YEARS  SMALLINT,
                             SALARY DECIMAL(7,2),
                             COMM   DECIMAL(7,2));
```

You can call **filltable()** to insert data rows now that the STAFF table has been created. The SQL data manipulation statments are in this external routine to allow them to be bound to the newly created database *after* the table has been created. Looking at FILLTB.SQC, you will see that it first binds itself to the database, then inserts the data rows. Binding after the table has been created allows all of the database object references in FILLTB.SQC to be checked against the actual table.

***Hints***

During your development, you must have a test database of the appropriate structure for precompilation of your programs. In the example given above, a database with the STAFF table (the IBM SAMPLE database) can be used for precompilation of the FILLTB.SQC program. You can easily create such test databases with the Query Manager. Since your database structure may change during development, using the Query Manager frees you from needing to modify and recompile a program to generate your test database.

## RETRIEVING DATA

Retrieving data is not as simple as merely executing a SELECT statement as you do when working with the Query Manager. Programs must handle data returned from a SELECT statement one row at a time. Imagine the set of rows described by a SELECT statement. Your program can move forward through this set of rows, retrieving each row's content for processing, one at a time. The Structured Query Language includes statements to help you access this set of rows.

### Retrieving a Single Row

If you know that a SELECT statement returns only one row of data, you can use the INTO clause to place the data values from that row into host variables. The following SELECT statements only return one row, since they both use SQL column functions to describe a single value based on all rows in the database:

```
EXEC SQL SELECT AVG(SALARY) FROM STAFF INTO :avg_salary;
EXEC SQL SELECT COUNT(*) FROM STAFF INTO :number_of_employees;
```

Most of the time you will be processing multiple rows of data rather than a single row. Even in the case of generating an average salary, you would most likely want a more meaningful number such as the average salary in each department (which would return one row for each department) rather than the average salary for the entire organization. You must use a different technique to retrieve a set of rows. The following section describes that technique.

### Retrieving Multiple Rows

Before working with a set of data rows, you must use the DECLARE CURSOR statement to describe the data with which you wish to work. This statement provides your program with a way to assign a name (called the cursor) to a SELECT statement. Here is the DECLARE CURSOR statement from the sample program DBSTRUCT.SQC.

```
EXEC SQL DECLARE current_row CURSOR FOR
         SELECT T.CREATOR, T.NAME, T.REMARKS,
                C.NAME, C.REMARKS, C.COLTYPE,
                C.LENGTH, C.NULLS, C.COLNO
                FROM SYSIBM.SYSTABLES T, SYSIBM.SYSCOLUMNS C
                WHERE T.NAME=C.TBNAME
                ORDER BY T.CREATOR, T.NAME, C.COLNO;
```

The SELECT statement describes a set of rows, often called the "results table," for which the cursor acts as a pointer.

After declaring the cursor, your program uses the cursor name (current_row in the example above) for all other manipulation. The first step is to OPEN the cursor. The OPEN statement is paired with the cursor name and readies Database Services to process the SELECT statement. For example:

```
EXEC SQL OPEN current_row;
EXEC SQL FETCH current_row into :tcreator,
                                :tname,
                                :tcomment :tcomment_nid,
                                :cname,
                                :ccomment :ccomment_nid,
                                :ctype,
                                :clength,
                                :cnull,
                                :colno;
```

Data is retrieved from the database with the SQL FETCH statement. When paired with the FETCH statement, the cursor serves as a pointer to the row of data to be retrieved. For every column in the select list, you must have a corresponding host variable (preceded with a colon) in the INTO clause of the FETCH to accept the value from the row. If the column being retrieved is nullable, you must specify a second host variable to hold the null indicator. Omitting a host variable for the null indicator can cause your program to terminate abnormally. Since specifying a null indicator variable on a non-nullable column does not cause an error, you should consider placing a null indicator on all columns.

### *DBSTRUCT.SQC*

The sample program DBSTRUCT.SQC is an example of how to retrieve a set of rows from a database. DBSTRUCT.SQC lists all the tables and their individual columns that exist in a requested database. This information is stored in the Catalog Tables, which are simply tables within a database that describe that database. You will want to run this program occasionally to review the data contained in your databases.

In looking at the SQL DECLARE SECTION, you see the host variables used to hold the data retrieved by DBSTRUCT.SQC. The comments on the right show the SQL column types that are FETCHed into the corresponding C variable. Notice that all of the character data columns have C character variables whose length is one greater than the

declared column length. This length ensures that the column data is FETCHed into null terminated strings for easy manipulation with the C string manipulation function calls.

This sample program follows the data retrieval model outlined in previous sections: open cursor, fetch data rows, close cursor. The sample program loops until no more data is found by the FETCH (outer loop) with the inner loop changing whenever a new table name is found.

DBSTRUCT.SQC is a general utility that you can run on any database. Of course, it must be bound to the database before it can execute any of its SQL statements. The following code fragment from DBSTRUCT.SQC shows how you can program such a utility to relieve the end user from having to do the bind. Notice that immediately after the first DML statement (OPEN CURSOR), DBSTRUCT.SQC checks the SQLCA for the return codes that indicate that the utility is currently unbound.

```
EXEC SQL OPEN current_row;

/* if this application is not bound to the database ...  */
if (sqlca.sqlcode == SQL_RC_E818 ||
sqlca.sqlcode == SQL_RC_E805)

  {
   step = "Bind Application to Database";
   sqlabind("dbstruct.bnd",argv[1],password,"",NULL,&sqlca);

   if (sqlca.sqlcode != SQL_RC_OK)/* error on rebind      */
      goto exit_error;
   else
      EXEC SQL OPEN current_row;  /* try to reopen the cursor */
  }

if (sqlca.sqlcode != SQL_RC_OK)   /* if error on opening cursor */
  goto exit_error;
```

The first condition checked (SQL_RC_E818) indicates that the current access plan is out of date (timestamp error). The program may have been bound to this database once before, but it has been modified and recompiled since then. The second condition (SQL_RC_E805) indicates that the access plan was not found. In this case, this is the first time this program has been run against the requested database.

In either case, DBSTRUCT.SQC then attempts to bind to the requested database. If successful, the original SQL statement (OPEN CURSOR) can then be executed again.

### *Hints*

The most serious limitation of using host variables (in any SQL statement, not just the FETCH) is the lack of support for C structures. "Moving Data To and From the Database" in Chapter 7 describes how to use pointers as host variables to place data into structures. Using this method, you can allocate a block of storage outside of the SQL DECLARE

SECTION, declare pointers to various portions of this storage, and use these pointers as host variables in the FETCH statement.

The code below is an example of how to declare host variable pointers and use them in a FETCH statement.

```
unsigned char buffer[30];          /* storage for row          */

EXEC SQL BEGIN DECLARE SECTION;

unsigned char (*first)[5];                /* for CHAR(5)       */
short         *second;                    /* for SMALLINT      */
short         *third;                     /* for SMALLINT      */
unsigned char (*fourth)[5];               /* for VARCHAR(5)    */
double        *fifth;                     /* for FLOAT         */
double        *sixth;                     /* for DECIMAL       */

EXEC SQL END DECLARE SECTION;

first  = buffer;                   /* initialize pointers      */
second = buffer+5;                 /* +5 for char[5]           */
third  = buffer+7;                 /* +5+2 for short           */
fourth = buffer+9;                 /* +5+2+2 for short         */
fifth  = buffer+14;                /* +5+2+2+5 for char[5]     */
sixth  = buffer+22;                /* +5+2+2+5+8 for double    */

EXEC SQL FETCH current_row INTO *first, *second, *third,
                                *fourth, *fifth, *sixth;
```

Note that the pointers must be initialized before the FETCH. Each pointer must be initialized to the proper offset into the buffer. The size of all prior variables must be accounted for when setting the pointer (that is, allocate 2 bytes for short data types, 8 for double, and so on). In the code the offset values are hard coded to show how the offsets are calculated. In actual practice you should use the C **length**() function to calculate the appropriate offset.

C statements that reference the data can be simplified by creating a union of the buffer with a structure that defines the individual data types. These C statements use structure references to the data rather than the buffer addresses.

Typically you FETCH character columns into null terminated C character arrays (thereby making it a string, which is easier to manipulate than an array of individual characters). However, the example in the code FETCHes character data into non-null terminated character arrays (array size = SQL column size). As in this example, you might wish to do this to construct a single character string from multiple SQL character columns. The buffer should be initialized to blanks before each FETCH to accommodate the varying length of VARCHAR data.

Do not use the SELECT * form of the SELECT statement in your program. Even though SELECT * provides a shortcut to specifying all the columns, you must still specify a host variable for each column being retrieved. Furthermore, using SELECT * unnec-

| SQL Data Type | FETCHed INTO . . or INSERTed from | Result |
|---|---|---|
| CHAR(n) | char[i] i < n | data truncated, not null terminated |
| VARCHAR(n) | char[i] i = n | not null terminated |
| | char[i] i > n | blank padded, null terminated |
| INTEGER | short | SMALLINT - ok, INTEGER - error |
| SMALLINT | long | ok |
| | float | ok |
| | double | ok |
| DECIMAL(p,s) | short | decimal portion truncated |
| | long | decimal portion truncated |
| | float | ok but loss of precision |
| | double | ok but loss of precision |
| FLOAT | short | decimal portion truncated |
| | long | decimal portion truncated |
| | float | ok but loss of precision |
| | double | ok |

**Table 8-1** SQL to C Data Conversions

essarily ties your program to the table structure. Explicitly listing the columns in the SELECT list allows additional columns to be added to the table without affecting your program.

Table 7-2 shows SQL data types and the matching C data types into which those columns are usually FETCHed. You can FETCH these columns into data types other than those and Database Services makes the appropriate data conversion. These conversions are limited to similar data types (that is, numeric SQL columns can only be FETCHed into numeric C variables and character into character). The conversions made are shown in Table 8-1.

Since IBM C/2 does not have a decimal data type, you use **double** to hold SQL DECIMAL data. Note that precision is lost in the automatic conversion to floating point data (which has at most 15–16 digits of precision, while the SQL DECIMAL can have 31 digits of precision). If your application requires the precision of DECIMAL data (for example, currency calculations in Italian lira), consider using dynamic SQL to retrieve the data in packed decimal format and do the conversion yourself.

The SQL FLOAT data type has double precision (8 byte floating point number with 15 significant decimal digits). If FETCHED into a **float** variable, precision may be lost.

DATE, TIME, and TIMESTAMP values are usually returned to your program in a format defined by the database country code, which is set from the OS/2 country code at the time the database is created. This is the LOCAL format. You can make your application independent of the LOCAL format by specifying one of the date/time format overrides during precompilation (see "Precompile and Bind" in Chapter 11 for information on what these formats are). On input, Database Services attempts to determine the date/time value

from the format of the input value. Database Services recognizes date/time values in the LOCAL, USA, European Standard, ISO, or Japanese Industrial Standard formats. Be aware that this flexibility can lead to some confusion. For example, consider a database whose LOCAL format is UK (DD/MM/YYYY). A date input in the USA format (MM/DD/YYYY) will either be accepted incorrectly or rejected due to an obvious conflict.

When FETCHing values based on SQL column functions, consider the type of data returned by the function. These data types are listed in Table 8-2.

## UPDATING DATA

There are two methods of changing data in your tables. The INSERT statement, which was discussed in "Initializing Table Data," can be used to create new rows of data. The other method is to change the values in a existing row of data. The SQL UPDATE statement is used for this purpose.

You can use the SQL UPDATE statement to change values in columns for a single row or a set of rows. Here are several examples of how to use the UPDATE statement:

- Update a single row, described by the employee number in the host variable **emp_no.**

```
EXEC SQL UPDATE staff SET JOB='PGMR'
                      WHERE ID = :emp_no;
```

- Update a set of rows described by a WHERE clause. If the WHERE clause is omitted, all rows in the table are updated.

```
EXEC SQL UPDATE staff SET JOB='CLERK'
                      WHERE ID IN (260058, 132341, 592361);
```

- Update the row pointed to by the cursor.

```
EXEC SQL UPDATE staff SET JOB='PGMR'
                      WHERE CURRENT OF current_row;
```

You use the first two methods of update when you know in advance what rows you wish to update. The other option is deciding to update the row you are currently processing while using a cursor. If you want to update with a cursor, you must first declare your intent to update specific columns when you declare the cursor. The columns to be updated must be listed in the FOR UPDATE clause of the DECLARE CURSOR statement.

Not all SELECT statements generate result tables that can be updated. Only one table can be specified and all columns of the SELECT must come from that table. In other words, you cannot update a join or any SELECT that includes a column function or arithmetic expression. The SELECT statement cannot include GROUP BY, ORDER BY, HAVING, or DISTINCT clauses. Finally, you cannot update a view that shares these same properties.

These restrictions limit the usefulness of UPDATE WHERE CURRENT OF CURSOR. A solution is to use a cursor to locate the row to be updated and then, using information from that row, use the UPDATE statement with a WHERE clause to change the specific row.

| Function | C Data Type | Notes |
|---|---|---|
| AVG | match column type used in function | NULL values ignored<br>NULL result if all columns NULL |
| COUNT | long | includes NULL values if COUNT(*)<br>excludes NULL values otherwise |
| MIN and MAX | match column type used in function | NULL values ignored<br>NULL result if all columns NULL |
| SUM | match numeric column type used in function | NULL values ignored<br>NULL result if all columns NULL |

**Table 8-2** C Data Types for Column Functions

In order to change a column to a null value, that column must be "nullable." That is, the column definition should not include the NOT NULL clause. When updating a nullable column you must supply two host variables for that single column. For example:

```
new_job_nid = -1;       /* cause column to be nulled */

EXEC SQL UPDATE staff SET JOB = :new_job :new_job_nid
         WHERE CURRENT OF current_row;
```

This updates the JOB column of the current row, setting it to a null value. You must also specify the null indicator when setting the column to a non-null value. The null indicator must be set to some value other than $-1$.

A column may also be set to a null value as follows:

```
EXEC SQL UPDATE staff SET JOB = NULL
         WHERE CURRENT OF current_row;
```

In this example the SQL keyword NULL achieves the same result as storing a $-1$ in the null indicator.

## DELETING DATA

The two ways of deleting data from tables both involve the SQL DELETE statement. The WHERE clause of the DELETE statement allows you to specify the criteria for the row(s) to be deleted. You can delete a specific row by specifying the columns that make up a unique key for the table, or you can delete a whole set of rows by specifying the minimum amount of information in the WHERE clause. For example, in the first statement below, a specific employee is deleted; the second statement deletes all employees who are programmers.

```
EXEC SQL DELETE FROM staff WHERE ID = :emp_no;
EXEC SQL DELETE FROM staff WHERE JOB='PGMR';
EXEC SQL DELETE FROM staff WHERE CURRENT OF current_row;
```

The third statement is used in conjunction with a cursor. You can delete the current row (the one just FETCHed) by using the WHERE CURRENT OF cursor clause with the DELETE statement.

## CONCLUSION

In this chapter you built a simple application program that connects to a database and creates, stores, and retrieves data with SQL statements. These elements are the most frequently used features of Database Services.

The SQL statements described in this chapter are limited because they must be coded within your program. As a result, you have no chance to change the type of SQL statement to be executed when your program is run. The next chapter describes a more flexible approach to including SQL statements in your database program.

# CHAPTER NINE

# Using Dynamic SQL

## OVERVIEW

In the previous chapter, SQL statements were placed directly in the program source. During execution, the SQL statement text (SQL verbs and object names) did not change. You will find some applications where you will want to change the SQL verbs and objects (for example, to execute the same query against a different table). Dynamic SQL provides a set of statements that lets you generate SQL statements and then execute them while your program is executing.

Writing programs to dynamically execute SQL statements poses a different challenge to the programmer than static SQL. In this chapter you will learn how the structure of a dynamic SQL program differs from one that uses static SQL. We guide you through the steps necessary to dynamically execute the different types of SQL statements. One of the sample programs, QUERY.SQC, illustrates the dynamic SQL concepts presented in this chapter.

## UNDERSTANDING DYNAMIC SQL

In the previous chapter you learned about static SQL. (This form of SQL is called static since the form of the statement is known at the time you code your program and does not change when the program is run.) An application that always executes the same SQL statements against the same database objects (such as an accounting report generator) is a good example of a static SQL application.

Unlike static SQL, the context of a dynamic SQL statement is not known at the time you code your program. As a result, you can change the statement every time you execute your program. A dynamic SQL application typically asks the user to supply the SQL statement or generate the SQL statement from some other type of input. The sample program, QUERY.SQC, puts up a simple prompt and expects the user to supply the entire SQL statement.

One reason you might choose dynamic rather than static SQL is when you do not know the names of any tables, views, or columns. (This will occur whenever you allow the user to name the tables and columns.) These names can be supplied by the user when your application is running and then built into dynamically executed statements.

Another reason for turning to dynamic SQL is if you do not know what types of queries are to be executed against your data. In this case, your program can dynamically build the

| Statement | Notes |
|---|---|
| BEGIN DECLARE SECTION<br>END DECLARE SECTION<br>INCLUDE<br>WHENEVER | Precompiler Directives |
| DECLARE CURSOR<br>OPEN CURSOR<br>FETCH<br>CLOSE CURSOR | Cursor manipulation statements |
| EXECUTE IMMEDIATE<br>PREPARE<br>DESCRIBE<br>EXECUTE | Dynamic SQL processing |
| SELECT . . INTO | Dynamic SELECTs are always processed with cursors |

**Table 9-1** Non-dynamic SQL statements

clauses to be used on your SQL statements (WHERE, GROUP BY, HAVING, ORDER BY). After building the complete statement text, it may then be executed dynamically.

Host variables allow static SQL programs to vary the conditions under which an SQL statement executes. For example, the following DELETE statement can be used to delete any employee whose number is in the host variable specified in the WHERE clause:

```
EXEC SQL DELETE FROM STAFF WHERE ID = :emp_no;
```

Dynamic SQL allows even greater flexibility because the SQL objects referenced in a statement (tables, views, columns) can be varied, as can any statement clauses (such as the conditions in a WHERE clause).

This additional flexibility does have its price. Each SQL statement executed dynamically must be compiled as well as have its access plan created *each time it is executed.* When you use static SQL, this work is done ahead of time when the program is being developed; using dynamic SQL, this work is done at the time you run your program.

The difference in performance is most noticeable if you perform a series of simple transactions (transactions with few SQL statements). The performance difference is more noticeable under these conditions because a SQL statement only needs to be compiled and the access plan created once per transaction. When a COMMIT or ROLLBACK is issued, the access plan is removed and must be re-created if the statement is to be re-executed. The only exception to this is a cursor that has been declared with the WITH HOLD option. The access plan for a SELECT statement associated with such a cursor will not be thrown away when a COMMIT or ROLLBACK is issued. See "Determining the Length of a Transaction" in Chapter 10 for more information on the WITH HOLD option.

Not all SQL statements can be executed dynamically. Table 9-1 lists those statements and the reasons they cannot be processed dynamically.

Follow these steps to dynamically execute an SQL statement:

1. **Generate the SQL statement text.**

   The statement you generate must be in a ASCIIZ character string. This host variable can be up to 32765 characters long and must contain only valid SQL characters (precompiler directives cannot appear).

2. **Determine the type of statement to be executed.**

   Your program must determine whether the statement to be executed is a SQL SELECT since SELECTs must be processed with a FETCH loop using a CURSOR; other statements can be executed immediately. These methods are discussed later.

3. **Execute the statement.**

4. **As with static SQL, check the SQLCA for return code information.**

## DYNAMIC STATEMENTS OTHER THAN SELECT

If you know ahead of time that your program will be processing statements other than SELECTs, you can simply capture the statement text into a host variable and pass it to Database Services on an EXECUTE IMMEDIATE statement. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
  char prep_string[2000];

EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :prep_string;
```

In one step, the EXECUTE IMMEDIATE statement checks the SQL statement for errors, creates the access plan, executes the statement, and removes the access plan.

Since the host variable, prep_string, cannot itself contain host variables, any data values in the dynamic statement must be explicitly specified. You can regain the flexibility that host variables provide through the use of parameter markers and the PREPARE and EXECUTE statements described below.

## INSERTING PARAMETER MARKERS

In order to use host variable substitution with dynamic SQL statements, place a question mark (?) in your generated statement where you would have placed a host variable. Thus, a statement that might appear statically as:

```
EXEC SQL DELETE FROM STAFF WHERE name = :lname;
```

would be placed into prep_string as:

```
DELETE FROM STAFF WHERE name = ?
```

A statement built in this way cannot be EXECUTEd IMMEDIATEly and must be processed in two steps. You must first PREPARE the statement for execution. The

PREPARE statement checks your SQL statement for errors and builds the access plan. In short, the PREPARE statement does for a single SQL statement what the precompiler/binder does for an entire program of static SQL statements. For example:

```
EXEC SQL PREPARE s1 FROM :prep_string;
```

The PREPARE statement assigns a name (s1, in the example above) to the prepared statement. This statement name is referenced in an EXECUTE statement, which actually performs the work. For example:

```
EXEC SQL EXECUTE s1 USING :lname;
```

Notice that the host variable is specified at this time. Parameter markers allow you to build and PREPARE the statement once then repeatedly execute the statement with different data. The overhead of checking the statement and creating the access plan is only incurred once.

## USING DYNAMIC SELECTS

SELECT statements pose a special challenge to the programmer who wishes to execute them dynamically—the programmer needs to understand what columns have been selected.

Your job is simplified if you know ahead of time what columns are being selected. However, in most cases you probably will not know. Since the SELECT statement is built dynamically, the column list may also be generated, possibly from user input.

The following sections discuss the two cases—*fixed list*, where the columns are known, and *varying list*, where the columns are unknown.

### Fixed List Selects

No matter what type of dynamic SELECT you are processing, you must still use cursors to retrieve the data. However, the way you DECLARE cursors is slightly different. With STATIC SQL, the DECLARE CURSOR statement actually contains the complete text of the SELECT statement. This method is insufficient for dynamic SQL since the DECLARE CURSOR (and therefore the SELECT associated with it) cannot be executed dynamically.

You might wonder how cursors get assigned to dynamic SELECTs. The PREPARE statement discussed earlier is used to assign a statement name to the SELECT, which is then later used with the DECLARE CURSOR. This technique is illustrated in the program fragment below.

```
EXEC SQL BEGIN DECLARE SECTION;
  char prep_string[2000];

EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE c1 CURSOR FOR s1;
```

```
main()
{
  read SELECT statement into prep_string

  EXEC SQL PREPARE s1 FROM :prep_string;

  EXEC SQL OPEN CURSOR c1;

  EXEC SQL FETCH c1 INTO host variables

  EXEC SQL CLOSE CURSOR c1;
}
```

This form of the DECLARE CURSOR statement serves only to assign a cursor name to a statement name. The precompiler does not generate any code from the DECLARE CURSOR. For this reason, a DECLARE CURSOR used with a dynamic statement can be placed anywhere in your program. In fact, a good practice with dynamic SQL is to place all of your cursor declarations in one place, preferably with your other declarations.

Once the statement is PREPAREd (and the cursor automatically assigned to it), you can begin retrieving rows of data as outlined in "Retrieving Data" in Chapter 8. Note that you are required to know how many and what type of columns are being FETCHed since you must supply the host variables on the FETCH statement.

The results of a dynamic SELECT can be altered with each execution through the use of parameter markers. Typically, parameter markers are used to vary the results of expressions in the SELECT statement, as can be seen in the following example:

```
SELECT NAME, DEPT, JOB FROM STAFF WHERE NAME = ?
```

Once this statement is PREPAREd you can execute it multiple times under different conditions simply by changing the host variable associated with the parameter marker. This host variable is supplied when the CURSOR is opened:

```
EXEC SQL OPEN c1 USING :emp;
```

***Note:*** With dynamic statements other than SELECT, parameter marker substitution is done on the EXECUTE statement. Since SELECT statements are not EXECUTEd, the host variables must be supplied on the OPEN CURSOR.

### Varying List Selects

You can build the greatest flexibility in your program when it makes no assumptions about the SELECT statement to be executed. In such a case, the columns to be SELECTed can vary with each PREPARE. But your program must discover both the number and the type of the columns to be retrieved. This information is provided to your program through a structure called the SQL Descriptor Area (SQLDA), in conjunction with the PREPARE statement.

### *The SQL Descriptor Area*

Use the following form of the PREPARE statement when you do not know what columns are being selected:

```
EXEC SQL PREPARE s1 INTO :sqlda FROM :prep_string;
```

Database Services analyzes the statement in prep_string and places its description in the SQLDA. Allocating space for your SQLDA is not as straightforward as the SQLCA since the SQLDA is not a fixed size; rather, its size depends on the number of columns being described. The SQLDA contains a fixed size header and an entry for each column being described as shown below.

```
struct sqlda
{
   unsigned char sqldaid[8];  /* Eye catcher = 'SQLDA   '       */
   long          sqldabc;     /* SQLDA size in bytes            */
   short         sqln;        /* # of SQLVAR entries            */
   short         sqld;        /* # of used SQLVAR elements      */
   struct sqlvar
   {
      short        sqltype;   /* Variable data type             */
      short        sqllen;    /* Variable data length           */
      unsigned char *sqldata; /* Pointer to variable data value*/
      short        *sqlind;   /* Pointer to Null indicator      */
      struct sqlnames         /* Variable name                  */
      {
            short          length;  /* Name length [1..30]      */
            unsigned char data[30]; /* Variable or Column Name */
      } sqlname;
   } sqlvar[1];
};
```

You can include the SQLDA structure definition directly in your program as follows:

```
#include <sqlda.h>
```

or you can let the precompiler generate the structure with the following statement:

```
EXEC SQL INCLUDE SQLDA;
```

Unlike the INCLUDE SQLCA statement, INCLUDE SQLDA does not allocate space for your structure. You must allocate the SQLDA yourself.

### *Allocating an SQLDA*

Determining the size of your SQLDA requires some thought. You could allocate the largest possible SQLDA, which can describe a maximum of 1489 columns and requires a 64k byte segment (this limit is derived from the 16 byte header plus 1489 44 byte variable descriptions), but this is probably a waste of space. A more economical approach is to dynamically allocate an SQLDA for the exact number of columns necessary. The following procedure is implemented in the sample program QUERY.SQC.

The first step is to use the PREPARE statement to obtain the number of columns in the dynamic SELECT. The SQLDA used with this first PREPARE can be the smallest possible SQLDA (16 bytes for the header). Database Services returns the number of columns in the sqld variable. You then use this value to allocate an SQLCA of the appropriate size.

The program fragment below allocates a minimum SQLDA. This fragment is more complicated than it need be for a 16-byte chunk of memory, but this code can be modified to allocate an SQLDA of any size.

```
static union {                  /* Pointer to FAR char string       */
               struct
               {
                unsigned offset;
                unsigned selector;
               } ptr4;
               char far * ptr;
             } segptr;

static struct sqlda far *sqlda = NULL;/* Pointer to an sqlda */
main()
{
    DOSALLOCSEG(SQLDASIZE(0),&segptr.ptr4.selector,PRIVATE);

    sqlda       = (struct sqlda far *)segptr.ptr;
    memset((char *)sqlda, '\0',SQLDASIZE(0));

    sqlda->sqln = 0;
}
```

Your code must complete each of these steps.

1. **Allocate space for your SQLDA.**

   Use the DOSALLOCSEG function to allocate memory in a separate segment to handle the large SQLDAs that your program may encounter. You can use the Database Services macro, SQLDASIZE, to return the requested SQLDA size in bytes. Just supply the number of variable entries required (zero in the figure). The SQLDASIZE macro is available to your program only if you use the INCLUDE SQLDA statement.

2. **Initialize the SQLDA.**

   The segment address returned by DOSALLOCSEG is stored as the address of the SQLDA. The storage is then initialized to zeros.

3. **Indicate how many variable entries have been allocated.**

   You are responsible for telling Database Services how many variable entries are in the SQLDA you use. Place this number in the sqln variable of the SQLDA header (0 in the figure).

Use this minimum SQLDA with the first PREPARE. When the PREPARE returns, use the sqld variable to allocate a new SQLDA of the appropriate size. You are now ready to get the statement description.

### *Obtaining the Statement Description*

Armed with a large enough SQLDA, you can rePREPARE the dynamic statement, but that would mean rechecking the syntax and rebuilding the access plan, which is not necessary the second time. Use the SQL DESCRIBE statement with this second SQLDA. The DESCRIBE statement describes the dynamic statement into the SQLDA without checking syntax or creating access plans.

```
EXEC SQL DESCRIBE :prep_string INTO :sqlda;
```

The variable entries now contain descriptions of the columns to be retrieved. The next section describes those entries and how to FETCH data from the dynamic SELECT.

### *Allocating Space for Row Data*

Before you can FETCH data rows from the dynamic SELECT, you must make sure you have enough space to hold the returned row. The description in the SQLDA provides enough information for you to do this.

Figure 9-1 shows a sample SQLDA after a DESCRIBE has placed column information into the variable entries. This example has only one column in the SELECT list (sqld variable is 1).

The names and meanings of the fields are listed below.

| NAME | MEANING |
|---|---|
| **SQLCAID** | An 8-byte identifier set to "SQLCA" and three blanks that act as an "eye-catcher" when examining computer memory. |
| **SQLDABC** | The total number of bytes in this SQLDA; 16-byte header plus one 44-byte variable entry. |
| **SQLN** | The maximum number of entries this SQLDA can hold. |
| **SQLD** | The number of column entries described in this SQLDA. |
| **SQLTYPE** | The SQL data type of the represented column. Table 9-2 shows |

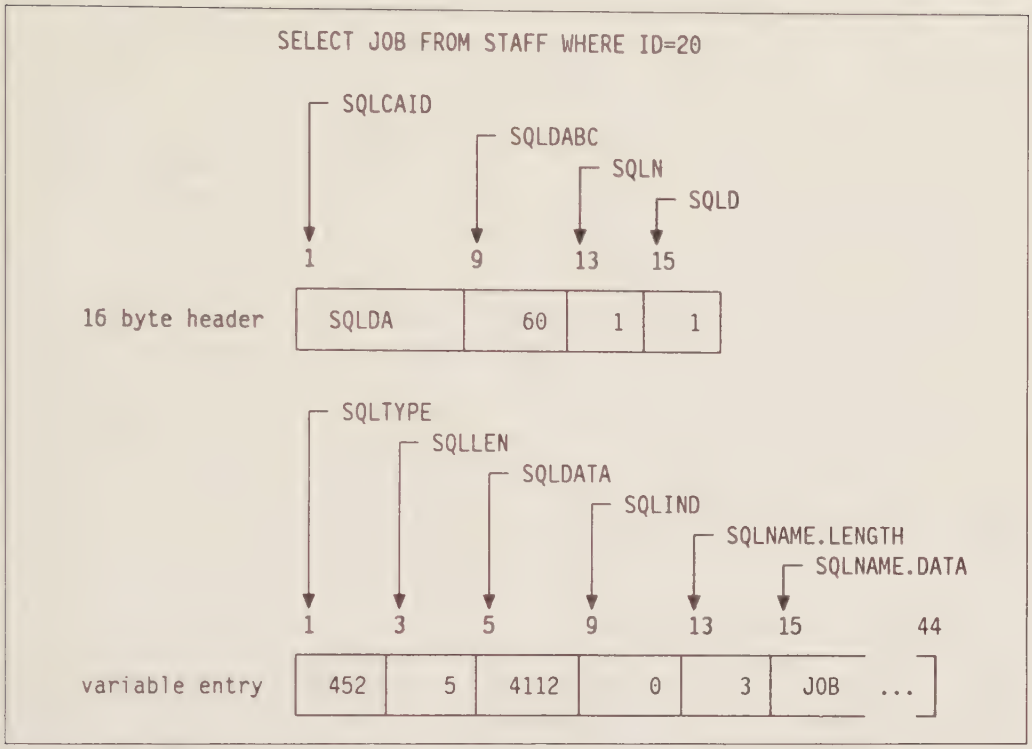



**Figure 9-1** SQLDA After DESCRIBE, One Column Described

all the possible SQL data type values. These are defined in the **sql.h** include file. This information, in conjunction with SQLLEN, tells you what type and how much space to allocate for this column data. Even values are non-null columns and do not require a null indicator.

**SQLLEN** The length of the column data.

**SQLDATA** Only has meaning for character columns. Contains the code page of the database containing the retrieved column. SQLDATA is zero when this column is defined FOR BIT DATA.

**SQLIND** Contains a 0 for character columns; undefined otherwise.

**SQLNAME.LENGTH** The length, in characters, of the column name.

**SQLNAME.DATA** The name of the column as it appears in the table definition. If the column is the result of an expression, this field contains an ASCII number representing the column's position in the SELECT list.

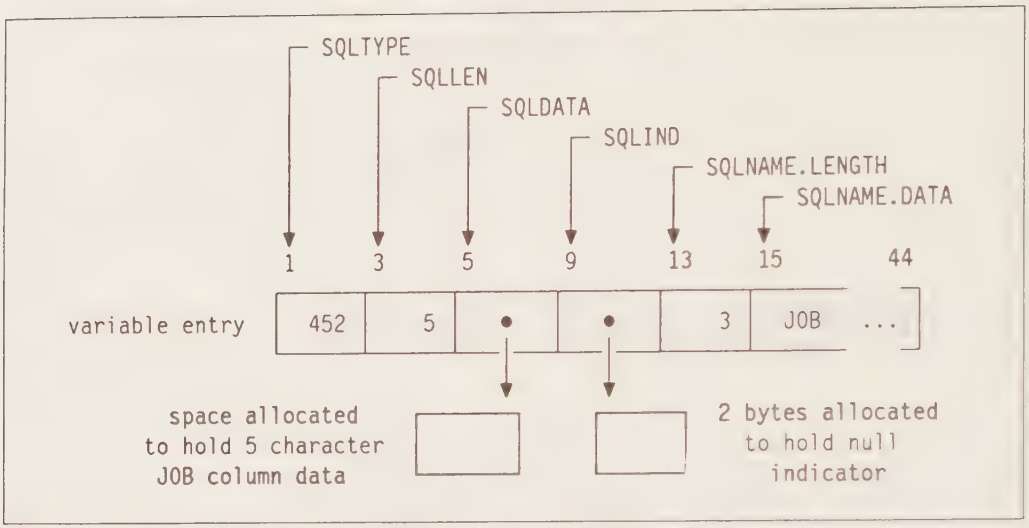



**Figure 9-2** SQLVAR Ready for FETCH

Your program must now scan the returned SQLDA, and for each variable entry (SQLD) allocate space to hold the column data and update the variable entry to include a pointer to this space. Figure 9-2 shows the variable entry from Figure 9-1 after it has been modified to point to the allocated storage. Store the address in the SQLDATA field. If the SQLTYPE value indicates that the column is nullable, you must also modify the SQLIND field to point to a 2 byte block of storage to hold the column's null indicator.

You can use the C **malloc** function to allocate space for each column to be returned. Simply supply the malloc function with the SQLLEN value from the variable entry and place the returned address in the SQLDATA field.

### *FETCHing the Row Data*

Once every variable entry has been updated, you can begin FETCHing the data. Use the following FETCH statement:

```
EXEC SQL FETCH c1 USING DESCRIPTOR :*sqlda;
```

Database Services reads the addresses from the SQLDA and places the retrieved data in the space you allocated. You can reuse the same space for the other rows or allocate new locations for each successive row (depending upon how much data you must keep).

Be sure to *cast* each of these data locations with the appropriate C data type. For example, the following should be used to reference a memory location allocated to hold an SQL FLOAT column, which was in the third position of the SELECT list:

| SQL Data Type | SQLTYPE | Notes |
|---|---|---|
| CHAR(n)<br>1 < = n < = 254 | 452<br>453 | Requires SQLLEN bytes. |
| DATE | 384<br>385 | Requires SQLLEN bytes (10). |
| DECIMAL(p,s) | 484<br>485 | Byte 1 of SQLLEN contains the precision, Byte 2 the scale. Allocate sufficient bytes for the precision. See hdref refid = cudnd. for information. |
| FLOAT | 480<br>481 | Requires SQLLEN bytes (8). |
| INTEGER | 496<br>497 | Requires SQLLEN bytes (4). |
| SMALLINT | 500<br>501 | Requires SQLLEN bytes (2). |
| TIME | 388<br>389 | Requires SQLLEN bytes (8). |
| TIMESTAMP | 392<br>393 | Requires SQLLEN bytes (26). |
| VARCHAR(n)<br>1 < = n < = 4000 | 448<br>449 | Requires SQLLEN + 2 bytes. After FETCH, the first two bytes of SQLDATA contain the length, the data begins at SQLDATA + 2. |
| LONG VARCHAR(n)<br>1 < = n < = 32700 | 456<br>457 | Requires SQLLEN + 2 bytes. After FETCH, the first two bytes of SQLDATA contain the length, the data begins at SQLDATA + 2. |

**Note:**
Of the two SQLTYPE values for each data type, the first (even) represents the non-nullable form. The second (odd) represents the nullable form.

**Table 9-2** SQL Data Type Values Returned by DESCRIBE

```
(double *)sqlda->sqlvar[3].sqldata     /* pointer to the data */
*((double *)sqlda->sqlvar[3].sqldata)  /* the data itself    */
```

## Data Types

Table 9-2 shows all the possible data types that can be returned by the DESCRIBE (or PREPARE INTO) statement. Notes are provided to help you understand the information returned in the SQLDA, particularly the SQLLEN value. There are two SQLTYPE values listed for each SQL data type, one each for the nullable and non-nullable forms.

The DESCRIBE statement faithfully reports the column data type as it exists in the table definition. Data conversions can be accomplished by changing the SQLTYPE value before doing the FETCH. Conversions may be desirable when handling varying character data or decimal columns (which have no corresponding C data type). Following sections address these conversions.

### *Processing Character Data*

Varying character data, including LONG varying character data, is always returned as a structure where the first 2 bytes contain the length of the character string. These 2 bytes are not included in the SQLLEN value. You must remember to allocate these 2 extra bytes when retrieving varying character data. For some languages (such as COBOL) this is the appropriate format for varying character data. The C language, on the other hand, does not have a special varying character data type. C uses null terminated character strings for varying character data.

You can get Database Services to return null terminated character strings by changing the SQLTYPE value before you FETCH the row. Change any VARCHAR (448/449) or LONG VARCHAR (456/457) to the C string data type (460/461) and Database Services returns a null terminated string. Note that the 460/461 SQL data type is never returned by Database Services. You place this value in the SQLDA prior to the FETCH to tell Database Services to convert the column into a ASCIIZ string.

Changing the fixed character data type (452/453) to the C string data type (460/461) allows fixed length character strings to be returned as null terminated strings.

You can simplify handling of DATE, TIME, and TIMESTAMP columns if you treat them as fixed length character strings. For these three column types and CHAR columns, the SQLLEN field contains the number of bytes to be FETCHed. All four column types can be processed by the same methods.

### *Processing Numeric Data*

Database Services automatically converts numeric data columns into the target type specified by SQLTYPE. For example, to convert an SQL DECIMAL column to FLOAT (so that the result can be manipulated in your C program) simply replace the DECIMAL SQLTYPE value (484, non-nullable or 485, nullable) with the FLOAT SQLTYPE value (480, non-nullable or 481, nullable). The sample program, QUERY.SQC, does this conversion just before allocating space for the column. The resulting data is a C **double** data value.

Some precision may be lost when Database Services performs this decimal conversion. The SQL DECIMAL data type can have up to 31 digits of precision. The FLOAT data type, on the other hand, has at most 15-16 digits of precision. If you require the precision offered by DECIMAL, you must provide your own math routines to operate on the SQL DECIMAL data type. You must also use dynamic SQL since it provides the only way of retrieving the actual packed decimal value.

To retrieve decimal data, you must not change the SQLTYPE field and allocate the correct amount of space for the packed decimal value. The SQLLEN field contains the

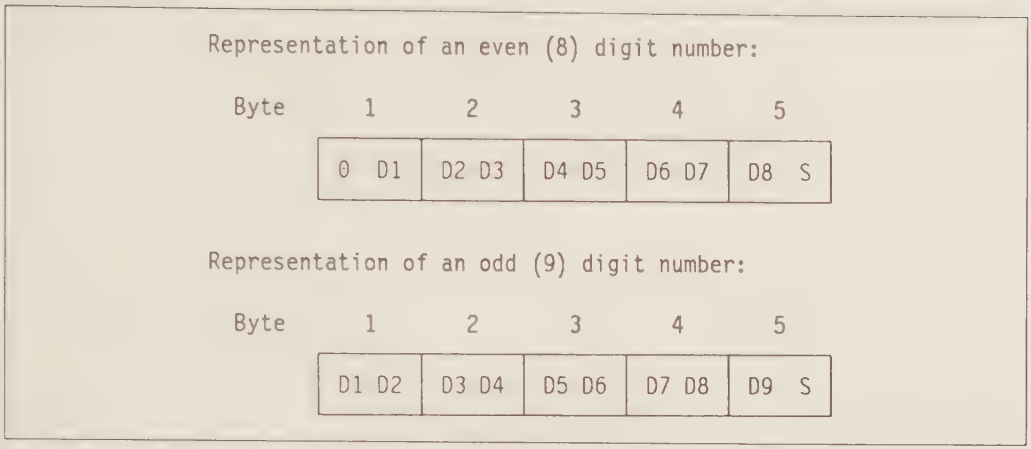

Representation of an even (8) digit number:
Byte 1 2 3 4 5
0 D1 D2 D3 D4 D5 D6 D7 D8 S
Representation of an odd (9) digit number:
Byte 1 2 3 4 5
D1 D2 D3 D4 D5 D6 D7 D8 D9 S

**Figure 9-3** SQL Data Type Values Returned by DESCRIBE

size of the packed decimal number, but cannot be used as with other data types. For DECIMAL columns, the SQLLEN field contains two numbers, the decimal precision in byte 1 and the scale in byte 2. The precision represents the total number of decimal digits and the scale represents the number of digits to the right of the decimal point.

In order to extract the precision and scale, you must understand one of the quirks of the PC's internal storage mechanisms. A word of random access memory is stored in reverse byte order. Thus, the location that is "byte 1" in physical order is actually "byte 2" in logical order. Consequently, IBM stores SQLLEN with the precision in physical byte 1 and the scale in physical byte 2, which means that they are reversed when you reference them from your C program. The following program fragment successfully extracts the precision and scale from SQLLEN:

```
int precision;
int scale;

precision = sqlda->sqlvar[1].sqllen & 0x00FF /* low byte */

scale = sqlda->sqlvar[1].sqllen >> 8       /* get high byte  */
```

To calculate the storage requirements for the decimal number, you must understand how decimal data is stored. Figure 9-3 shows that decimal data is stored in a packed format—two decimal digits per byte of storage. An extra half byte is required to hold the sign; if there are an even number of decimal digits, a "pad" digit is added in the most significant byte. Database Services always rounds a DECIMAL column's precision to odd. By doing so, Database Services guarantees that a "pad" digit is not necessary.

The following equation returns the correct number of bytes to hold the decimal number:

```
bytes = (precision+2)/2
```

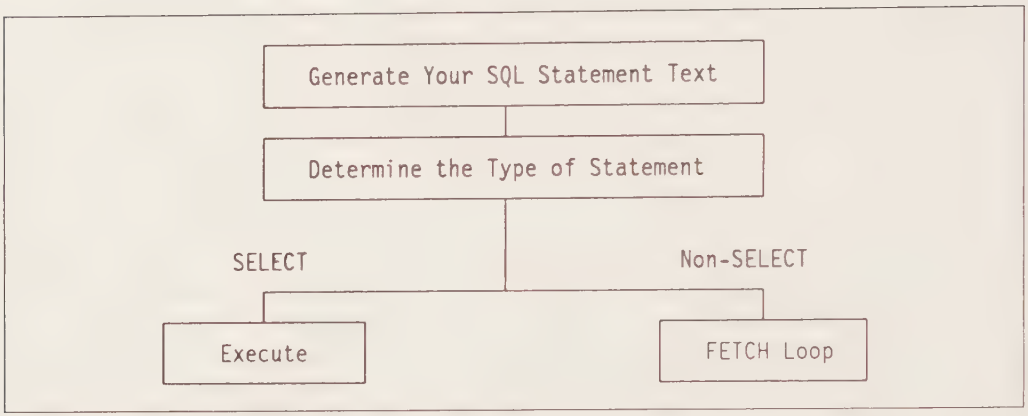



**Figure 9-4** Application Program Model for Dynamic SQL

## BUILDING A SAMPLE PROGRAM

Up to now, the methods described have involved prior knowledge of the type of SQL statement to be executed. How do you build a program to execute *any* SQL statement that may come its way? Figure 9-4 shows the program structure required to handle any type of SQL statement.

There are several steps in this general structure:

1. Generate the SQL statement to be executed.
2. Determine the type of SQL statement to be executed.
3. Execute non-SELECT statements

   These statements can be executed simply by using the dynamic SQL EXECUTE statement.

4. Process SELECT statements.

   SELECT statements are processed in the same manner as static SELECT statements, using the FETCH statement within a loop.

The only step that has not yet been explained is how to determine the type of SQL statement. You need not scan the statement yourself. The SQL PREPARE statement returns a piece of useful information, the number of columns in the SELECT statement. If this number (in the sqld variable of the SQLDA header) is zero, the statement is *not* a SELECT.

The sample program, QUERY.SQC, is an excellent example of how to construct a

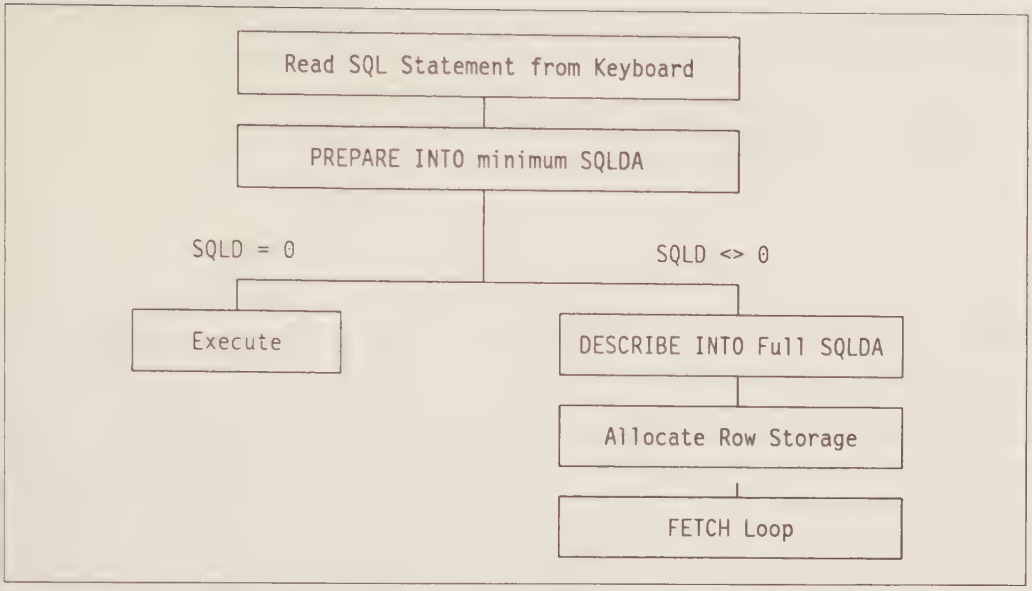



**Figure 9-5** QUERY.SQC—Sample Program Structure

program for dynamic execution of any SQL statement. Figure 9-5 shows the logical structure of QUERY.SQC. This figure reflects the basic structure shown in Figure 9-4; only specific implementation details have been added.

QUERY.SQC employs most of the techniques discussed in this chapter. Most dynamic SQL applications have logic very similar to QUERY.SQC. Modify this program to fit your application needs and you are well on your way to completing your dynamic application.

## HINTS

With dynamic SQL, the DECLARE CURSOR statement's sole purpose is to assign a cursor name to a prepared statement name. For example:

```
EXEC SQL DECLARE c1 CURSOR FOR s1;
```

Since the DECLARE CURSOR cannot be executed dynamically, the cursor names you assign are fixed at the time you write your application. For this reason, you should consider the number of PREPAREd SELECT statements that you want to keep active.

Your application can build (and PREPARE) a number of SELECT statements and have them "ready to go" when requested. However, since the DECLARE CURSOR statement cannot be executed dynamically, you must also have the cursors for those SELECT statements DECLARED and "ready to OPEN." Therefore, for each SELECT you expect to have ready, place a DECLARE CURSOR in your program. Since these DECLARE

CURSOR state ments do not produce any executable code, you can place them all in the declaration section of your program. For example:

```
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL DECLARE c2 CURSOR FOR s2;
EXEC SQL DECLARE c3 CURSOR FOR s3;
   .
   .
   .
EXEC SQL DECLARE c20 CURSOR FOR s20;
```

***Note:*** You can only have 8 cursors OPEN at the same time.

## CONCLUSION

Dynamic SQL is one of the most powerful tools available to the database programmer. The flexibility provided by dynamic SQL allows you to write general database programs that can operate on many different database structures. The sample program, QUERY.SQC, is a perfect example of how flexible a dynamic SQL program can be. This sample program can operate on any database. Notice, however, that the decision of what operations to perform is left to the user of QUERY.SQC. Flexibility has its price.

The first three chapters of Part 3 of this book have concentrated on the most frequently used function calls and SQL statements. Although you might be tempted to put this book down and finish coding your own database program, you should spend some time considering the effects that a database system might have on your programming techniques. The next chapter covers these database programming considerations.

# CHAPTER TEN

# Designing Your Database Program

## OVERVIEW

In Part 2, "Database Application Design Considerations," we discussed application design considerations unique to data bases: database structure, consistency, and configuration. In the first three chapters of this part, we have shown the fundamentals of using SQL and the Database Services function calls in a database program. This chapter ties all of this information together and discusses the final programming decisions you must make to complete the actual coding of your own database program.

The general model for database applications was outlined in Chapter 7, "Database Program Fundamentals." This model was explained in simple terms as start using a database, do your work, and then stop using the database. This chapter expands upon that original model.

Special attention is paid to the process structure of database applications handling some of the more serious events such as deadlocks. Special sections at the end of the chapter outline where you might use static versus dynamic SQL and pose some challenges to those who might like to create their own interface to Database Services.

Do not overlook the *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference*, which also contains a chapter of useful tips for the database application designer.

## USING THE DATABASE APPLICATION PROCESS MODEL

The basic elements of an OS/2 multitasking application are *processes* and *threads*. The relationship between these elements and your database access is important to understand.

Whenever your program makes a request to Database Services (either a SQL statement or a function call), that request is processed as part of your process. In other words, all calls to Database Services are synchronous (that is, each must complete before your application receives control back from Database Services). If you need to do some work while a request is being processed, you can use other OS/2 threads or processes to achieve the necessary concurrency yourself.

Under OS/2, operating system resources are owned by a process. These resources are shared among all the execution threads of that process. Database resources are no exception. An OS/2 process can have at most one database open at a time. You can access dif-

ferent databases serially (for example, open/close database A and then open database B), but you cannot reference tables from both databases simultaneously. For this reason, you should place all the data your application needs into a single database. If you have multiple applications that share common data, place both the common data and each application's private data into the same database. Placing all data into one common database avoids having to spawn a child process to access a secondary database. You should avoid this since SQL SELECT statements cannot reference tables (that is, do a join) from more than one database.

Within a process, you can create individual threads of execution in order to multitask your application. Each of these threads can use SQL statements to access tables in the one open database (the database is a resource owned by the process). Even though multiple threads can concurrently execute SQL statements, there is only one transaction for the process. Therefore, if any single thread executes a COMMIT or ROLLBACK, *all* work done by *all* threads is COMMITed or rolled back. The transaction is therefore made up of the work done by all threads of the process.

***Note:*** Even though it appears to your program that SQL statements in multiple threads execute concurrently, Database Services serializes them and only executes one at a time per process.

## WORKING WITH ONE DATABASE

In "How Many Databases?" in Chapter 6, we gave several reasons why you should only have one database per OS/2 volume. In this section you will learn how your database program can accommodate this situation.

To allow a user to maintain a single database per OS/2 volume, your program should not automatically create its own database. The best approach is to ask the user at installation for a database in which your application can place its data. Your application should be prepared to create a database in case the user has not already created one. The database name provided by the user can be saved in a file as part of your application. When you start your application, read the database name from the file and use that name when your program connects to the database.

This approach works well for applications that maintain their own data, such as payroll or inventory applications. However, there are some classes of applications called "utilities" that can run against multiple databases. The sample program DBSTRUCT.SQC, for example, displays the name and column structure of all the tables in the requested database. In this case, your program should ask the user for a database name each time it is executed.

You need to choose an appropriate qualifier for your application's objects if they are to be placed in a user's database rather in a database created solely for your application. Be careful to avoid possible naming conflicts with other objects that may already exist in the user's database. You may wish to use your company's name or a portion of it as a (you hope) unique qualifier. Another alternative is to ask the user to supply an appropriate qualifier. The qualifier can be saved in the same file as the user's database name.

## DATA HANDLING

The set orientation of SQL can affect the methods that your application uses in handling data. Unless you specify the ORDER BY clause, the results rows returned by a SQL SELECT statement have no order implied. This lack of physical ordering is different from some PC data manager systems. For example, dBASE associates a physical record number with each row of data. You can use this record number to access a specific row. The Database Manager (as well as most SQL database products) does not associate a record number with each row of the table. With SQL, the way to reference one and only one row is by specifying a primary key on the WHERE clause of a SELECT statement.

Also note that there is no "end of table" maintained as in some file management systems. If you are familiar with these other types of systems, you will have to stop yourself from positioning to the "end of the table" before adding data. With the Database Manager, data is INSERTed into a particular table and you need not worry about the physical location of the data.

At the programming level, you retrieve data from tables one row at a time. Working with one row at a time isolates your program from another physical aspect of your data, the storage required to hold the rows returned by the SELECT. If your application needs to work with more than one row of data simultaneously, it needs to buffer the rows required for its operation.

Programs use cursors to move through the rows of a table. These cursors move in only one direction, forward. For this reason, be sure to specify the ordering you wish with the ORDER BY clause on the SELECT statement. Forward-moving cursors can present problems if you need to scroll backward through your data. Buffering rows allows your application to scroll backwards through rows already retrieved, but you may have too many rows for your application to buffer. If the table has a unique key field, your application can remember this key from a previous row. If a backwards scroll is required, use the remembered key to reposition the cursor at the earlier position in the table. Your application can then process forward from that point.

If you need to process a table in two different orders at the same time, you can declare two cursors. One should be ordered ascending on the required key, and the other descending on the same key.

## MANAGING TRANSACTIONS

Once your application starts using a database, a transaction is automatically started and continues until it issues a COMMIT or ROLLBACK, then another transaction is automatically started, and so on, until your application stops using the database. How you manage these transactions affects your application design. The awareness of and ability to control transactions can be passed through to the user(s) of your application, or they can be maintained inside the application. A correct assessment of transaction duration versus user action is a key design choice in a concurrent application.

This section discusses some of these application design considerations. "Transaction Overview" in Chapter 4 contains further information on transactions.

### Determining the Length of a Transaction

Do not write your application assuming that it is the only database program executing. OS/2 is multitasking and, while only a single user can be running a machine, database servers bring the reality of multiuser databases to the PC. For this reason, you should give serious consideration to the length of a transaction.

The length of a transaction determines the length of time that your application holds locks on rows, thereby preventing other applications from sharing that data. As a general rule, the shorter your transaction the better. Keeping transactions short requires different techniques depending on whether your application is batch oriented or interactive.

With batch oriented applications, human interaction is not required once the program has started. The length of your transaction is determined by the placement of COMMITs and ROLLBACKS within your program's logic. On the other hand, with interactive applications COMMIT points are usually dictated by some action taken by the user.

If you have your application wait for user input while holding locks, you adversely affect other applications by increasing the chance that those applications may request data that your program is holding. For this reason, you should minimize the locks your program holds while waiting for the user to take some action.

The HOLD option of the DECLARE CURSOR statement can help you reduce the locks your application holds. In an interactive application, your program can FETCH as many rows as necessary for display, then COMMIT the transaction to release the locks. Without the HOLD option, the COMMIT also closes the cursor and you lose your position in the results table. The HOLD option allows the locks to be released, but leaves your cursor open and ready to FETCH the next row. Since the locks are released, other applications can continue to update any rows necessary. When your user finally chooses an action, reread the requested row to verify that it has not changed since it was first read.

This method allows the previously read row to change (or even be deleted) before the user selects it for update. If you do not want the row to change between the time it is presented to the user and selected for update, you should minimize the data read for display. This will decrease the number of locks your transaction holds while waiting for the user to select an action.

No matter how well behaved your transaction management is, your application can still be the victim of someone else's poor transaction management. Consequently, your application must be prepared to handle waits and deadlocks. The following sections discuss how to handle these two situations.

### Deciding When to Acquire Locks

In "How Locking Works" in Chapter 4, we said that you can explicitly control locks only on tables. Rows may be automatically locked when your transaction references them. In this section, we discuss some actions you can take to influence when Database Services acquires locks on behalf of your transaction and, therefore, influence the amount of concurrency.

To increase concurrency, you can delay OPENing CURSORS until you actually need them. For example, if your transaction did not read table T until late in the transaction,

then table T would be available to other transactions for update purposes during most of the interval that the original transaction was inflight. By designing your application in this manner, you can increase the number of transactions referencing the same object, but you also increase the chance of encountering a wait. If your transaction decides to UPDATE a row read by another transaction, it must wait until the reading transaction completes. These waits can impact the performance of your application.

If you choose to design for performance of your single application rather than concurrency, you should consider OPENing all your CURSORs at the front of your program (especially ones DECLAREd FOR UPDATE). This design is good since some conversion deadlocks can be avoided and your transaction has a reduced chance of hanging in the middle while waiting for a conversion.

### Handling Waits

Tables or rows locked exclusive by other applications can prevent your application from updating rows read by other applications and reading rows updated by other applications. If your application attempts to access a row that has been locked exclusively by another transaction, your program waits, indefinitely.

If all goes well, the application holding the lock will COMMIT or ROLLBACK in order to release the lock. If the offending application does not COMMIT or ROLLBACK, you must abort your transaction and retry. You must use OS/2 multithread support or a Ctrl+Break signal handler to interrupt a thread waiting on a lock.

If your application is interactive, you can let the end user decide when to interrupt the transaction. Create a Ctrl+Break signal handler that asks the user whether the current transaction should be aborted or whether the entire application should be stopped. Use the Database Services interrupt function (**sqlaintr**) to abort the current transaction. The user decides whether it should be retried.

A batch application must handle interruption of waits in a different manner. Since there is no end user involved, your application must decide for itself when to abort a waited transaction. You can use the multitasking support of OS/2 to create a separate thread to time how long a particular SQL statement is processing. If a specified amount of time passes before the statement completes, this thread can issue the Database Services interrupt function to abort the transaction.

Of course, you can combine these two methods in an interactive application that prompts the user after a SQL statement has been processing for a specified period of time. The user can then choose to continue or abort the transaction.

### Handling Deadlocks

As "Deadlocks" in Chapter 4 pointed out, no matter how well behaved your application is, it can still be the victim of a deadlock. If Database Services detects a deadlock, it selects one of the deadlocked transactions as a victim. That transaction receives the SQL -911 deadlock detected error indicating that it has been rolled back. Your application can choose to handle this error through a graceful recovery or by terminating the program. In order to perform a graceful recovery, your program must keep some sort of state information to

know how much work was rolled back. The transaction that was rolled back can then be re-attempted.

The information kept by your application should record the COMMIT points of your transaction. This information can then be examined to determine how much work had been COMMITed before the error. Your application can then pick up where it left off and complete in a normal fashion.

### Handling Log Full Conditions

When your transaction encounters the log file full condition (SQL error code -964), it can either COMMIT or ROLLBACK. If there are multiple concurrent transactions accessing this database, the log full condition may be temporary. These other transactions may COMMIT and free up enough space in the log file to allow your transaction to continue. For this reason, you should consider retrying the operation that received the log full error code. If you constantly run into this problem, either your transaction is too long or you should increase the size of the database's log file.

## DESIGNING THE USER INTERFACE

Using SQL to manage your data frees you from some of the programming chores of accessing data, thus allowing you to spend more time providing meaningful interfaces to your application.

The use of SQL presents some interesting choices for user interfaces. A very simple application (such as the sample program QUERY) presents SQL itself as the user interface. On the other extreme, your application can build a graphics interface with the Presentation Manager and totally hide the fact that the data is stored in a relational database.

The following classification of user interfaces may help you decide what type of interface is best for your application. These interfaces are listed in order of increasing isolation from the SQL interface.

1. **Interactive SQL** This kind of interface allows the user to enter and execute an interactive SQL statement directly. The Query Manager SQL Query window is an example of this kind of interface. Your user must be experienced with SQL in order to use your interface. This type of interface requires minimal programming (see QUERY.SQC).

2. **Assisted Interactive SQL** In this type of interface, an interactive SQL statement is built, but the interface assists the user in the construction of that statement. The Query Manager Prompted Query is an example of this kind of interface. Depending on the complexity of the assistance, the user may not need to have any knowledge of SQL. Of course, more programming is required, but your handling of SQL is the same as in number 1 above.

3. **Add-in SQL** The name of this type of interface is taken from the Lotus 1-2-3 add-in concept. In this kind of interface, additional functions and menu items are added to an existing product in order to allow the user to issue SQL queries and bring the data into the data management environment provided by the product. The add-in is clearly a

secondary method of data access to the primary one provided by the product. As an example, a SQL add-in in Lotus 1-2-3 might bring relational data into an area of a spreadsheet so that it could become part of the entire spreadsheet calculation. However, the primary data object would still be the spreadsheet with SQL database access in a supporting role. This type of interface is primarily used for establishing data interchange between existing products and the Database Manager. Note that the same function could be implemented via export and import.

4. **Non-SQL data management** This class of interface has a wide variety of data managers, including both relational and non-relational variants. The key point here is that the functions offered to the user are translated into the appropriate SQL statements without the user's knowledge (the user does not see SQL directly). In this kind of interface, the user accesses the Database Manager through the primary functions provided by the interface; in the add-in interface, the user clearly knows that alternative functions are being used to access a secondary SQL database. Paradox SQL Access, DataEase, and dBase IV 1.1 have stated that they intend to put their user functions on top of the Database Manager platform.

5. **Specialized function** This type of interface is created for a specific purpose, with functions that hide the SQL used underneath to access the data. An example might be a special application a chain store creates for point-of-sale transactions.

## BINDING'S EFFECT ON PROGRAM DESIGN

The database objects referenced in your static SQL statements must exist at the time you bind your program to a database. This condition affects your development process and the structure of your application. Figure 10-1 shows the following methods to structure your application.

**Method 1** As part of your application development, build your database and bind your program to it. You then copy this database to whatever machine will run your application. The database backup and restore utilities can be used to move a database from one machine to another. You can actually avoid programming these steps by having the user perform them through Query Manager.

**Method 2** Build a program that creates your database objects. Run this program as part of your application installation and have it bind the other parts of your application (DOWORK.SQC, the data manipulation).

Because of its flexibility, we recommend Method 2. See the sample programs INSTALL.SQC and FILLTB.SQC in Appendix B for an example of how to implement this method. The advantages of Method 2 are listed below.

1. **Provides end user flexibility** You can ask the user which database should be used (allows the user to manage all data within a single database). In this situation, be sure

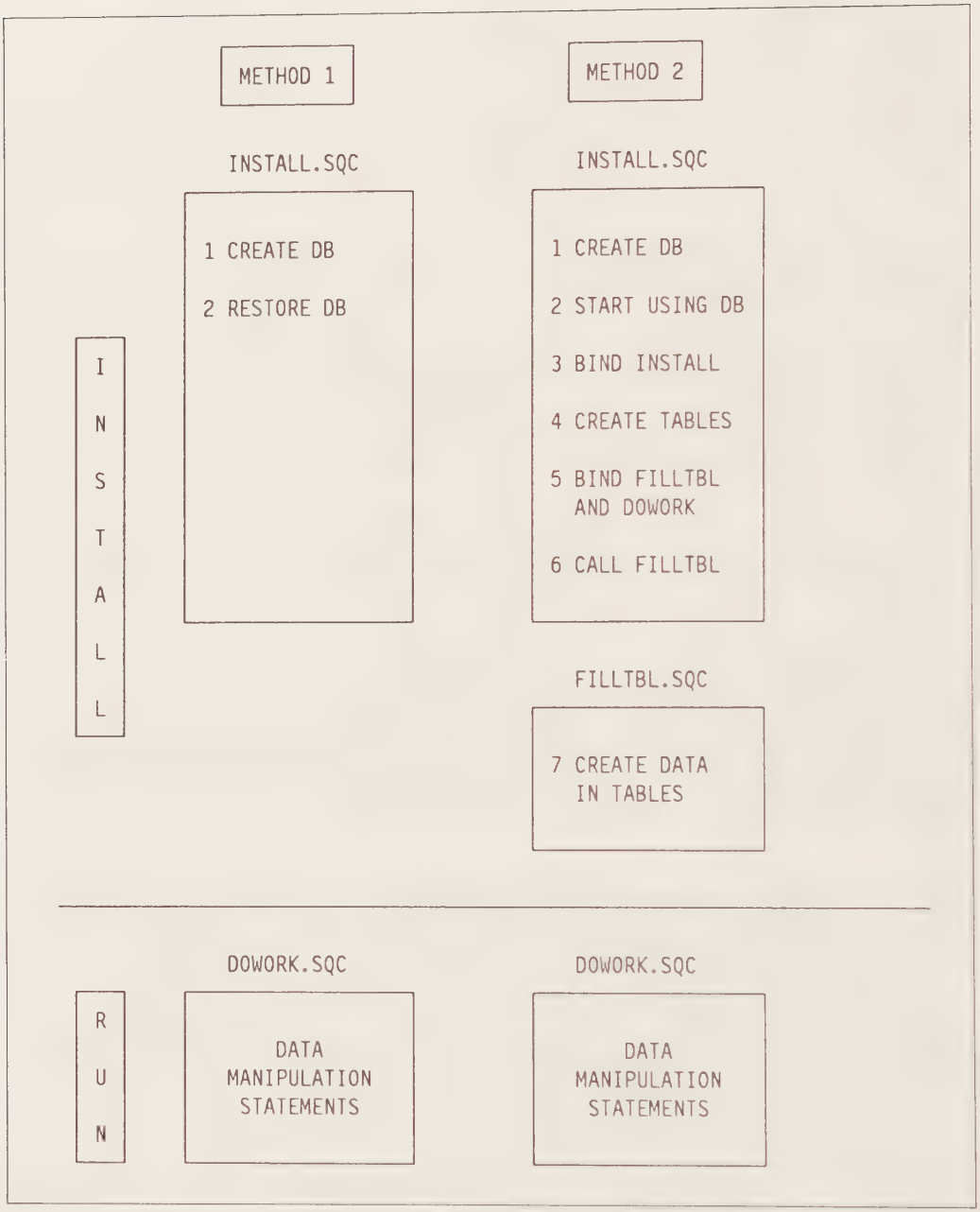



**Figure 10-1** Effect of Bind on Application Structure

to use a qualifier to qualify your object names (tables, views, indexes) to prevent collision with other application objects.

2. **Allows data sharing with other applications** By placing your objects in one database specified by the user (see above), your application can easily use data from other applications (since SQL statements can only be issued against objects from a single database).

3. **Simplifies providing new versions of your application** By building your database at installation, rather than restoring an existing database, you achieve flexibility in providing updates to your application. If your application requires a large amount of installed data, you can provide table updates with import/export rather than providing a whole new database.

Design your database objects before writing your program. Create these objects and try some of your planned queries (the Query Manager provides a handy way to do this). When you have a structure (tables and views), you can create a program to build that structure. This program can either create the database or use an existing one and then bind itself to that database (steps 2 and 3 in Figure 10-1) just before creating the structure. When you precompile this program, use the test database you built earlier, but delay the binding so that the installation program can do the bind.

If your application requires data to be loaded into the database, you can load the table as part of the installation program. FILLTB.SQC loads the new table by executing a series of SQL INSERT statements. This program must also be bound to the database prior to execution, but it must be bound *after* the table it references has been created; this is the reason the INSERT statements are in a separate program from INSTALL.SQC. FILLTB.SQC can be precompiled in a separate step, and a bind file created specifically for the INSERT statements. In this manner, INSTALL.SQC can be bound before the table is created and FILLTB.SQC can be bound after the table is created.

A more convenient method for loading your database tables is with the import utility. A program using import does not require binding. If you use the IXF format, your installation program need not create the tables during installation. The installation process only requires database creation, data import, and binding of the application itself.

## CONCLUSION

Take the time to consider the topics presented in this chapter before you finish coding your database program. The time you spend now to design these database system features will save you problem-solving time later.

You should now be ready to move on to the mechanics of precompiling, compiling, and linking your database program. The next chapter shows how you perform these steps to convert your program into an executable application.

# CHAPTER ELEVEN

# Precompiling, Compiling, and Linking

## OVERVIEW

You will now convert convert your program source into a running database application. In this chapter you will learn how to precompile, compile, bind, and link your database program. If you have never worked with a database system before, you should pay particular attention to the discussion of the precompile and bind steps. These two steps can affect how you design your application. The other steps necessary to make an executable program from your source files (compiling, linking, and running) are also explained.

## PROGRAMS YOU WILL NEED

This chapter assumes that you have the following programs:

1. **The IBM OS/2 Extended Edition Version 1.1** After installing the base operating system, you must install the Database Manager Database Services.
2. **The IBM C/2 Compiler Version 1.1** You must also install the IBM C/2 compiler. Install the Large Model and Alternate Math Libraries. These libraries are required by the Database Services libraries you use in your database application.
3. **The IBM OS/2 Programmer's Toolkit Version 1.1** Also install the OS/2 Programmer's Toolkit. There are many features you can install with the Toolkit. These are the recommended options for developing C language database applications:
   - **C Language Samples** If you are unfamiliar with C, these sample programs may help you get started with C.
   - **Presentation Manager Samples** You must install these samples if you plan to use the Presentation Manager with your database applications. The Presentation Manager include files and resource compiler are placed on your computer.

After installing the C compiler, modify the SET PATH command in your CONFIG.SYS to reference the C compiler BIN directory, which contains the compiler commands. Installation creates a command file, in the BIN directory, to set up the environment variables for the compiler (LIB, INCLUDE, TMP). This command file is named NEW-VARS.CMD.

NEW-VARS sets LIB, INCLUDE, and TMP to the appropriate values for the C compiler. In order to use the C compiler with Database Services functions, you must modify these environment variables to include the Database Manager and OS/2 Programmer's Toolkit directories.

The command file shown below was used to prepare all of the sample programs in this book.

```
@ECHO OFF
@SETLOCAL
If %1. == . goto NOPARM
If %2. == . goto NOPARM

If exist %1 goto MSG1
If exist %1.SQC goto DOIT
Echo You Goofed! No "SQC" file by this name.
Goto END

:DOIT
call new-vars
set lib=c:\sqllib;c:\os2;%LIB%
set include=c:\sqllib;j:\toolkt11\c\include;%INCLUDE%

sqlprep %1.sqc %2 %3 /b
if ERRORLEVEL 1 goto MSG2

cc %1.c /AL /W2 /DLINT_ARGS;
if ERRORLEVEL 1 goto MSG2

link %1,,,sql_stat+sql_dyn+llibcp+doscalls;
Goto END

:NOPARM
Echo You goofed! Specify: file_name database [password]
Goto END

:MSG1
Echo You goofed! Specify input file name without extension.
Goto End

:MSG2
Echo You goofed! Errors on Precompile or Compile.
Goto End

:END
@ENDLOCAL
```

The only requirement is that the C\BIN directory must be in the PATH statement (which you should have done when you installed the C compiler). NEW-VARS is executed to set up the C compiler environment. The SET statements modify the appropriate envi-

ronment variables to include the new directories. NEW-VARS includes the \OS2 directory in the LIB environment string. If you plan to do any compilations from a drive other than C: (where \OS2 resides) you must either change NEW-VARS to set LIB to C:\OS2 or add it in your own batch file (as shown above).

## THE PRECOMPILATION PROCESS

When placed in your source file, SQL statements must be converted to function calls that your language compiler understands. This conversion process of precompilation converts the embedded SQL statements into calls in the source language.

Precompilation also creates the access plan. Access plans contain information on how your SQL statement are to be processed. For example, an access plan might contain instructions regarding which index to use when an SQL SELECT statement is executed. A separate access plan must be created for each source file that contains SQL statements.

Access plans are further subdivided into *sections*, which contain the plan for a particular SQL statement from the source file. The function calls created during precompilation include a parameter that allows the appropriate section of the access plan to be located when your application is executed. Figure 11-1 shows the relationship between your source file, the Database Services API, and the access plan for that source file.

Your source module (containing SQL statements) must have a .SQC file extension. This extension identifies the file as a C language program that contains SQL statements. Your program could have the normal extension .C if it did not contain any SQL statements; therefore, programs that use only the Database Services function calls can have the .C extension.

You should remember, however, that you can make a procedural simplification by having all of your database programs use the .SQC extension, and therefore be processed by the precompiler (this is mentioned in "The SQL Communications Area" in Chapter 7). There is then no need to distinguish between those programs that require precompilation and programs that do not—they all do!

## PRECOMPILE AND BIND

After completing your source file, you must PRE-compile the .SQC file. Precompilation, using the IBM supplied program SQLPREP, performs several important functions:

- Checks the syntax of your SQL statements
- Converts your SQL statements into the "host" language (in this case, IBM C/2)
- Checks the semantics of your SQL statements
- Creates an access plan

These steps are similar to those performed by a program language compiler. Semantic checks require further discussion since they place some requirements on the precompilation process. These semantic checks perform these functions:

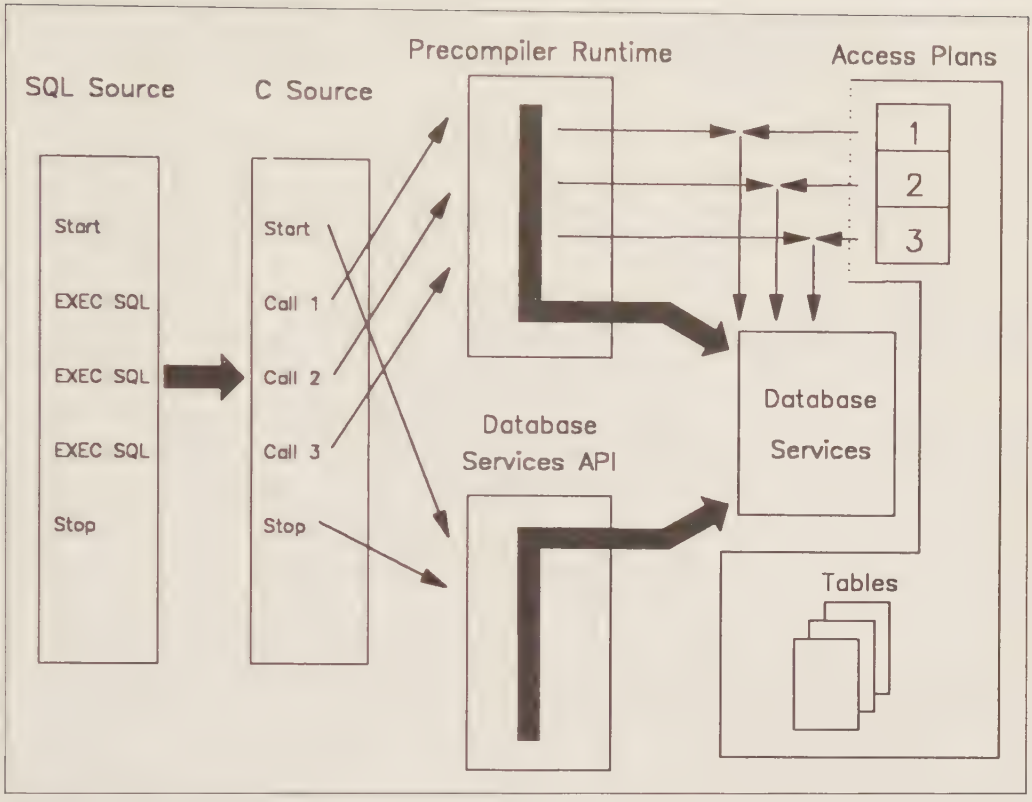



**Figure 11-1** Precompilation and Access Plans

- Verify that the table names and columns referenced in your SQL statements actually exist. (Therefore, a database with these objects must exist before you precompile).
- Verify that columns in the SELECT list can be appropriately converted into the host variable specified on the FETCH. For instance, an SQL DECIMAL column can be FETCHed into a C double variable, but the precompiler flags an error if you attempt to FETCH a DECIMAL into an unsigned character array.

In order to give the precompiler the name of a database with which to make these checks you must create a "test" database to perform the precompilation. Although you could write a program to create your database, the Query Manager provides an easier way to create tables for this purpose. You can also use the Query Manager to modify the table's structure. This feature is very handy since your table structure may change frequently during your development.

Since the database must exist before precompilation, you need to follow a specific design methodology for developing your database application. You must first complete your

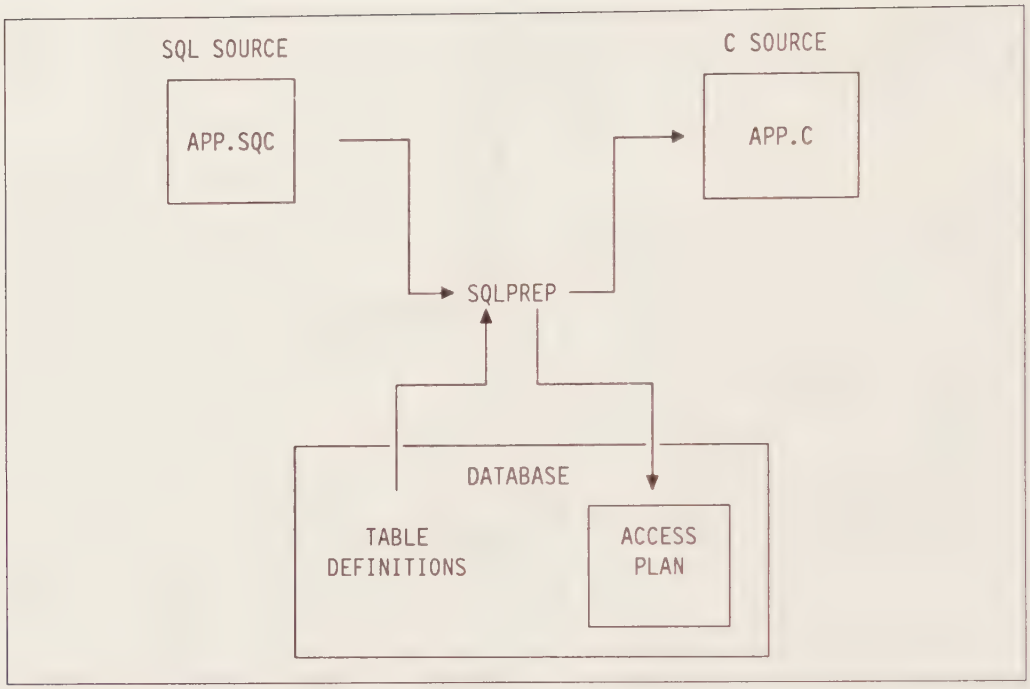



**Figure 11-2** Precompilation Access Plan Created

database design before designing your program. This method follows the classic design model of deciding the inputs and outputs *before* designing the process to perform the function. You may find it very difficult to begin your program if you have not decided what your table definitions are!

*Note:* You need an existing database structure (tables, views, indexes) only when precompiling static SQL programs. Since an application composed entirely of dynamic SQL makes no references to particular database objects, those references cannot be verified. However, the precompiler always requires a database to do its work. If no objects are referenced from your program, you can use any database during precompilation.

### Precompile With Bind

The output from the precompiler is represented in Figure 11-2. Notice that the precompiler generates the access plan and stores it in the database. This process of creating an access plan is called binding your program to the database. This binding can be delayed and done in a separate step discussed in the next section, "Precompile Without Bind."

Here is the precompilation step used in "Building a Sample Database Program" in Chapter 7:

```
[C:\TESTPGM]sqlprep tutorial.sqc sample
```

This command precompiles the SQL program TUTORIAL.SQC against the IBM SAMPLE database, to which TUTORIAL.SQC specifically connects. SQLPREP has the following form:

```
sqlprep program_name database [password]
```

No password is specified if the database does not require one. SQLPREP accepts the following optional command switches:

1. **/B <file name>** Do not create an access plan for this program. Instead, create a file (the *bind file*) from which an access plan can be created at a later time. See "Precompile Without Bind" in this chapter for more information.

2. **/F <date/time format specification>** Overrides the default date and time formats returned to your program by Database Services. Using this parameter guarantees what format is used in returning date time values to your application. You can request the following formats:

   - USA (U.S. Standard) mm/dd/yyyy hh:mm AM or PM
   - EUR (European Standard) dd.mm.yyyy hh.mm.ss
   - ISO (International Standards Organization) yyyy-mm-dd hh.mm.ss
   - JIS (Japanese Industrial Standard) yyyy-mm-dd hh:mm:ss
   - LOC (Local format) The date and time formats are determined from the database country code, which is determined from the OS/2 country code in effect at the time the database is created.

3. **/M <message file name>** Specifies a file to contain the processing messages generated by the precompiler.

4. **/P <access plan name>** Specifies the name for the access plan. If not specified, the program name is used as the name of the access plan. This name is used in the source code generated by the precompiler. The name is passed to Database Services to locate the access plan in the System Catalogs.

5. **/S** Forces the precompiler to perform only a syntax check. No bind file or access plan is created.

The precompiler does not expand macros or include files, which prevents you from placing SQL statements in those types of files. Placing SQL statements into include files is desirable if you have multiple source files in your application referring to the same database tables. In this situation, it would be nice if you could define host variables for this table once and refer to them as extern in all other files by including a single file that contains the SQL DECLARE section for these extern host variables.

You can work around the restriction by using the C compiler to generate a preprocessor listing. The C preprocessor expands macros and include files and generates a new source file that can be passed to the SQL precompiler. The precompiler output can then be

processed as before. Here is an example of how to use the IBM C/2 preprocessor in conjunction with the SQL precompiler. Remember to specify any required preprocessor options (such as the define or include options):

```
cc program.prp /E /DLINT_ARGS; >program.sqc
sqlprep program.sqc database password /b
```

The precompiler creates the C source file that you pass to the language compiler. This file has the same name as the .SQC file but with an extension of .C. Each of your SQL statements is placed in the output file as C language comments and zero or more C language function calls to Database Services. Do not be surprised that a SQL statement may not generate any code. Some statements, such as WHENEVER, tell the precompiler what to do with *other* SQL statements. These type of statements are called precompiler *directives*. The BEGIN and END DECLARE SECTION statements are also directives. These SQL statements tell the precompiler that the enclosed variable definitions are used with other SQL statements.

Since the number of C statements varies per SQL statement, predicting the amount of code growth is difficult. In general, each SQL statement generates at least one C call to perform the function, with one additional C call for each host variable in that SQL statement.

You can use C comments (/* */) within your SQL statements. However, since the IBM C/2 compiler only honors one level of comments, you should use the SQL comment delimiter (--) in order to avoid problems if you need to comment out an SQL statement. The precompiler turns your SQL statement into comments in the generated C file, but it avoids the nested comment problem by converting any C comment delimiters into some other character string so the C compiler doesn't notice them.

### Precompile Without Bind

You need not bind your program to a database when you precompile. If you specify the /B switch, SQLPREP creates a bind file. This bind file can be used at a later time to build the access plan. IBM provides the SQLBIND command to create an access plan from a bind file.

Delaying the binding step is desirable in the following cases:

- You wish to delay binding until you have a program that compiles successfully. There is no need to take the extra time to do the bind until you have a program that is ready to execute.
- You want to create the access plan under a different, possibly even multiple, qualifiers. You can change the qualifier for an access plan by changing the value of the SQLUSER string before executing the bind.
- You wish to run your program against many different databases. By delaying the bind, you can precompile your program once and bind it to multiple databases. Since precompilation is not required when you want to run your program against a new database, you do not need your program source code present (.SQC file) when executing.

- You wish to run your program against the same database on many different machines. If you have a program that always runs against the same database, you might think you could bind your program once. This is true, but you would then have to install that exact database on every machine that is to run your program (that database contains the access plan). By delaying the binding, you can dynamically create your application database on each machine and then bind your program to the newly created database as part of your application installation process.

TUTORIAL.SQC is a fairly generic program. It operates on the system catalogs, which are present in every database. With a simple modification, it can be made to work on any database by accepting the database name from the console instead of hard coding it in the program. With TUTORIAL.SQC written in this manner, you should delay the binding until you know which database to run it against.

The name of the bind file is taken from the name of the program being precompiled. The file extension is .BND. You may optionally specify a new name for the bind file, immediately following the /b. Assuming that the /b option is used when this modified TUTORIAL.SQC was precompiled, the following command accomplishes the bind to another database:

```
[C:\TESTPGM]sqlbind tutorial.bnd otherdb
```

SQLBIND accepts two command switches:

1. **/M <message file name>** Specifies a file to contain the processing messages generated by the binder.
2. **/F <date/time format specification>** Overrides the default date and time formats returned to your program by Database Services. See the description above of SQLPREP for a list of supported formats.

The end user need not be aware that your application must be bound to a particular database. Depending on the type of application you have written, the bind can be done either as part of your installation process or when the user actually invokes your program. This capability is facilitated by the availability of a function call to the bind utility **sqlabind.** The following program fragment was taken from the sample program DBIMPORT.SQC. This code shows how DBIMPORT.SQC uses the **sqlabind** utility to bind itself to the database requested when DBIMPORT was invoked (contained in the database variable).

```
/************************************************************/
/* Bind ourselves to the database                           */
/************************************************************/

sqlabind("dbimport.bnd",database,password,"",NULL,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)
   goto exit_error;
```

If your application comes with or creates its own database, you can bind all of your application to this database immediately after that database has been created. If your application is a general purpose utility, you can issue the bind function call when the user invokes it against a particular database. TUTORIAL.SQC, in Chapter 7, is such a program.

During your program development you may forget to rebind your program after making a correction. When you attempt to run this program, Database Services detects a timestamp error because the timestamp of the access plan in the data base is not the same as the timestamp associated with the executable program. Simply rebind before running your program again.

## COMPILING

After precompilation, the C source file must be compiled. Use the following IBM C/2 compiler invocation to compile the C source file output by the precompiler. This same command was used to compile all of the sample programs presented in this book. The case of each parameter is significant.

```
cc program.c /AL /W2 /DLINT_ARGS;
```

1. **/AL** You must compile your database application with the LARGE model since the Database Services were compiled with that model (you can also use the HUGE model if you find the need). Compiling with the LARGE model defaults your procedure calls and pointers to the appropriate size (far calls and 4 byte pointers).

2. **/W2** This parameter is optional but recommended. W2 causes the compiler to generate message for normal warnings and automatic data conversion warnings.

3. **/DLINT_ARGS** This parameter generates function call prototypes for type checking. The Database Services include files honor the LINT_ARGS identifier in the same manner as the C compiler include files. By defining LINT_ARGS, the C compiler enables type checking on the Database Services function calls.

4. The C compiler supports other options that you may find useful.

   - **Math Library** You can choose whichever math library you desire for compilation. The IBM C/2 compiler defaults to use the Emulator library. Specify the library with the /FPx options, where x indicates the specific math library you desire. Since Database Services itself was compiled with the C alternate math library routines it cannot take advantage of a math coprocessor if present.
   - **Optimizing** Database applications can be optimized with the same compiler options you use with other applications. The /Ox option maximizes the optimization provided by the C compiler.
   - **Debugging** If you intend to use a symbolic debugger with your program, you need to use compiler options to include line number (/Zd), symbolic debugging information (/Zi), and disable optimization (/Od).

The *IBM C/2 Compile, Link, and Run* contains complete information on all of the compiler options.

The Database Services routines use PASCAL call/return sequences (left to right), which are different from default C language sequences (right to left). The Database Services include files override the C default call/return sequence by declaring the Database Services routines with the **pascal** keyword. Your own routines can still use the C language call/return sequence.

Locating errors after the compilation can be a small problem since the compiler reports errors against statement line numbers from the C file, not the SQC file. Unless you have a tremendously good memory or a very short file, you must look in the generated C file to see what is wrong with your C statement. **Do not change the statement in the C file.** Instead, change it in the SQC file and execute the precompiler before recompiling.

Forgetting the semicolon at the end of your C statement is one of the more irritating errors to locate since the compiler usually reports an obscure error on a following statement. When compiling output from a precompiler, this situation becomes more confusing since these errors may be reported on statements that the precompiler generated. As a result, you must look in the precompiler generated code and work backwards in order to locate the statement in error.

## LINKING

Use the OS/2 Linkage Editor to link your object modules with the required C, OS/2, and Database Services libraries. These required libraries are

- LLIBCP.LIB—C language, large model, protect mode library.
- DOSCALLS.LIB—OS/2 link library.
- SQL_STAT.LIB—Database Manager static link library.
- SQL_DYN.LIB—Database Manager dynamic link library.

The IBM C/2 compiler includes a command to compile *and* link your program in one step (cl command). The following OS/2 command file uses the cl command to compile and link the output from the SQL precompiler:

```
sqlprep %1.SQC %2 %3 /b
cl /AL /W2 %1.c sql_stat+sql_dyn+llibcp+doscalls
```

This is a handy command file to use in your program development.

## DEBUGGING

If you wish to use a symbolic debugger with your program, be sure to specify the proper compiler options. See "Compiling" in this chapter for more information.

The precompilation process can complicate your use of symbolic debuggers. These symbolic debuggers work with the C language statements generated by the precompiler, not

the SQL statements that you embed. For this reason you may wish to test your SQL statements *before* you place them in a C program.

You can build and test your database and SQL statements in the Query Manager SQL Query Window before doing any programming. The Query Manager can be used in the following ways:

1. Create the tables and views to be used with your application
2. Fill these tables with test data using the Query Manager import command, SQL INSERT commands entered in the SQL Query Window, or with the **Add Data Rows** option
3. Test the SQL statements your application executes using the SQL Query Window
4. Create queries through the Prompted Query panels and then convert them to SQL in order to learn the statements that must be used in your application program. This might be useful if you are unfamiliar with SQL.

## CONCLUSION

You should now be able to write and convert into a running program your database application. Even more important, you should be aware of the impact that the precompile and bind steps have on your database program design. We have also shown you the advantages of delaying the bind step until after your program is precompiled.

The next chapter is the final one on building database programs. In it you learn how to use some of the Utility and Environment functions.

# CHAPTER TWELVE

# Utility and Environment Functions

## OVERVIEW

Storing and retrieving data is only part of a complete application. You must also provide certain housekeeping functions for exchanging or protecting data and improving performance. This chapter describes these functions.

The functions for database maintenance and administration come from three Database Services categories—the Environment, Utility, and Configuration routines. These routines are accessed via C language function calls. The following functions are described:

- Improving Performance—Reorganize and Run Statistics
- Exchanging Data—Import/Export
- Environment—Create and Delete Database, Catalog Manipulation, and Configuration

These functions will be covered in order of importance to your database program. Some of the less frequently used functions, such as exchanging data and the environment functions, can simply be performed from the Query Manager rather than programmed in your application. The section discussing each function recommends where you might benefit by supplementing your application with the Query Manager function rather than programming it yourself.

The sample programs described in this chapter provide some convenient functions to be used during the development of your own program. You can use the sample program function calls as models for your own application. Also, you can use them to build your test databases and import test data, once you have compiled the sample programs.

The backup and restore function calls are intentionally not covered. You should not need to program these functions unless you are providing an alternate user interface to Database Services. For most applications, it is sufficient to direct the user to the Query Manager in order to back up and restore databases. This is an especially wise recommendation if your application stores its data within a database shared by multiple applications.

## CODING THE UTILITY AND ENVIRONMENT ROUTINES

This section discusses some of the coding features common to all of the routines discussed in following sections. These common elements include the way the Utility functions do their work and the program parameters unique to the routines that move bulk data to and

from your database. You need not be connected to a database in order to use any of the Utility routines. The Utility routines create a separate OS/2 process in which they perform any necessary database operations. If your program is connected to the database on which you wish to perform the utility operation, be sure that you have no outstanding transactions against that database. An outstanding transaction would likely cause a deadlock where the separate utility process is waiting for your calling process to complete. To avoid this deadlock, remember to issue either a COMMIT or ROLLBACK before calling the utility.

Some of the Utility routines deal with moving data to and from databases. These routines are

- Backup
- Restore
- Import
- Export
- Reorganize Table

The likely source/target of information for these routines is diskettes. Each utility manages the space on the diskette so you need not worry about it. If multiple diskettes are required, the utility returns with 3180 in the SQLCA SQLCODE field. Your application can then prompt the user for the next diskette. In this manner, your application can control the external appearance of the utility (this is important if you are using the Presentation Manager). After the user has placed the next diskette in the drive, your application must call the same utility again.

Use the **callerac** parameter to tell the utility that this second call is a continuation of the operation started on the previous call. Set **callerac** to zero the first time you call a utility routine. Subsequent calls should set **callerac** to 1 to continue processing. A value of 2 terminates the utility.

The database backup and restore utilities, while accepting a **callerac** parameter, do not return to your program for diskette handling. These utilities call the OS/2 backup and restore commands to perform the operation. These commands already handle the swapping of diskettes.

## IMPROVING PERFORMANCE

There are certain maintenance activities that your application can do to increase performance. This section discusses the performance-related Database Services functions reorganize and run statistics, along with the configuration API.

You should include the reorganize and run statistics functions as part of your database program. These functions require intimate knowledge of the tables and indexes used by your application. These details can be hidden from your user if your program chooses the tables and indexes, as well as the point in time, to reorganize and update statistics. On

the other hand, there is no reason to program the configuration routines. You should rarely need to change the configuration parameters.

Database statistics include such items as the number of rows in a table and the overall size of a table. Statistics are important because Database Services chooses the optimal access strategy based upon the values stored in the Catalog Tables. The statistics can influence the choice of whether an index or table scan is used to find a particular row in a table. That choice will affect performance.

The statistics are not updated automatically by Database Services. The only time the statistics are updated is when the statistics utility is invoked. Application designers (or database administrators) must use the statistics utility periodically to ensure proper optimization of data access.

You should run statistics in the following situations:

- After executing a large number of updates, deletions, or insertions
- After reorganizing a table
- After creating or updating a table via an import
- After creating a new index

DBIMPORT.SQC, discussed in "Exchanging Data with Import and Export" in this chapter, embodies some of the performance techniques described in this section.

### Reorganizing Your Database

A database that has undergone many deletes has empty space in both the table and associated indexes. Use the reorganize utility to improve performance by reclaiming this space. Reorganization is provided through the **sqlureor** Database Services function call.

```
sqlureor(database,password,table_name,index_name,
         temp_file, callerac,&sqlca);
```

All parameters are pointers to ASCIIZ strings except for **callerac** and **&sqlca.**

Reorganization is accomplished via an export and a subsequent import of the table data. This can result in a time-consuming operation and require a large amount of disk space. If the volume containing the database has limited space available, consider specifying the temporary reorganization file on a different volume.

No processes can be using the database at the time your application requests a reorganization. The reorganize utility, which runs in its own process, requests exclusive use of the database in order to perform the reorganization.

This restriction makes reorganization a difficult task to add to the normal flow of your application. Instead, it should be run when the user notices diminished performance. Your application can keep track of the number of updates since the last reorganization and recommend when the user should reorganize.

You should regenerate statistics for a table after reorganization. Statistics generation is covered in the next section.

### Running Statistics

The Catalog Tables (discussed in "Catalog Tables" in Chapter 3) contain statistics on tables and indexes. These statistics are used by the optimizer to build the access plans for your application, but they are not automatically kept up to date. Your application must use the Run Statistics function call, **sqlustat**, to generate these statistics.

The sample program DBIMPORT.SQC issues the Run Statistics function call after the data has been imported and the user has been given a chance to create indexes on the table. Here is the function call from DBIMPORT.SQC:

```
sqlustat(table_name,0,null,SQL_STATS_BOTH,SQL_STATS_CHG,&sqlca);
```

The meaning of the parameters is as follows:

1. The first parameter, table_name, is a pointer to an ASCIIZ string containing the fully qualified table name (qualifier plus name).
2. The second parameter, 0, indicates how many indexes are to have their statistics generated. The zero indicates that all indexes are to be calculated.
3. The third parameter, null, is a pointer to a list of indexes for which statistics will be generated. If all indexes are to be used (parameter 2 is zero), this parameter should be set to null. The only reason you might want to select the specific indexes is to reduce the length of time it takes to run statistics since statistics can take a while to run on large tables and indexes.
4. The fourth parameter, SQL_STATS_BOTH, indicates that statistics are to be generated on both the table and its indexes.
5. The fifth parameter, SQL_STATS_CHG, indicates that DBIMPORT.SQC allows other programs to change the tables and indexes while statistics are being generated. You can alternately restrict other users to read-only access. In a standalone environment, this parameter does not make much difference. In a server environment, which is likely to have a greater amount of concurrent operations, you should allow others to read and change the table while statistics are being generated. The statistics may be slightly out of date as soon as they are calculated.

You must rebind any static SQL applications that use this table in order to take advantage of the new statistics. You can use the **sqlabind** function call to execute the bind immediately following the run statistics. See "Precompile Without Bind" in Chapter 11 for an example of how to use **sqlabind.**

## EXCHANGING DATA WITH IMPORT AND EXPORT

Database Services provides function calls for data import (**sqluimp**) and export (**sqluexp**). Use these routines to exchange data with other applications (data in another format).

The following data exchange formats are supported by Database Services:

- IXF—Integration eXchange Format
- DEL—Delimited ASCII
- WSF—Lotus Work Sheet Formats (WKS, WK1, WRK, WR1)
- ASC—Fixed ASCII (non-delimited) for Import only

These formats are described in "Export" in Chapter 16. They are also described in the discussion regarding import export from the Query Manager. The ease of performing import and export from the Query Manager should make you evaluate your need to program these functions yourself. If your user wishes to exchange data with another application, direct the user to the Query Manager to perform the necessary function. Details of the import and export functions provided here will help you build an alternate interface to Database Services or if you need to use the import function to initially load your database.

The import and export function calls are discussed together because they both accept the same parameters. Here is a sample call to the import function:

```
sqluimp(database,password,datafile,dcoldata,tcolstrg,
        filetype,filetmod,msgfile,callerac,&sqlca);
```

The name and meaning of each parameter in the call above follow:

| NAME | MEANING |
|---|---|
| **database** | Pointer to database name |
| **password** | Pointer to database password |
| **datafile** | Pointer to the path and name of the OS/2 data file |
| **dcoldata** | Pointer to structure describing how to use the OS/2 data |
| **tcolstrg** | Pointer to structure describing how to use the target table |
| **filetype** | Pointer to name of data file format |
| **filetmod** | Pointer to structure containing format specific options |
| **msgfile** | Pointer to path and name of OS/2 file for error messages |
| **callerac** | Access parameter |
| **&sqlca** | The address of an SQLCA |

Following sections explain how to use the structures required for import and export. The program fragments are taken from the sample program DBIMPORT.SQC, which appears in Appendix B.

### Specifying the Data Format

The **filetype** parameter is simply a pointer to an ASCIIZ string containing one of the valid supported formats—IXF, DEL, WSF, ASC. Note that the *IBM OS/2 Extended Edition Database Manager Programming Guide and Reference* lists the IXF format as "PC/IXF". This name indicates that the Database Manager IXF format is different from the 370/IXF format supported by the IBM Query Management Facility (QMF). However, you must specify "IXF" as the **filetype** string on the import/export.

These four strings do not supply sufficient information about the data file to allow the operation to complete (for example, it will not indicate which Lotus format is being used). For this reason, an additional structure is required.

The **filetmod** parameter is a pointer to a structure that describes further characteristics of the data file. The actual contents of this structure depend upon the format specified in **filetype.** Table 12-1 shows the values the **filetmod** parameter can assume.

The **filetmod** structure is a Database Services varying-length character string. These strings are made up of a length followed by the character data itself. You can use the **sqlchar** structure (declared in the **sqlutil.h** file) to allocate space for **filetmod** as follows:

```
struct sqlchar *filetmod;       /* pointer to filetmod        */
filetmod = (struct sqlchar)malloc(length)
```

If you use the **sqlchar** structure, you must dynamically allocate the space since the structure definition is only for a character string of length 1. Choose a length great enough to hold the largest string your program will generate.

### Specifying Columns

Next, you must describe the columns that are in the data file and the target table. The **dcoldata** parameter is a pointer to a structure that describes the columns in the OS/2 data file. The **tcolstrg** parameter describes the target table in the database. These parameters are used differently depending on the direction of the exchange.

#### *Columns for Import*

For import, the **tcolstrg** structure contains information on which database columns the data is to be imported. This structure is a Database Services varying length character string (**sqlchar** structure) containing the following text string:

```
{INSERT | REPLACE | CREATE | REPLACE_CREATE} INTO table[(column list)]
```

where

- INSERT means that the data is to be added to the specified table.
- REPLACE means that the original data is to be replaced by the import data.

| Data Format | filetmod Values | Notes |
|---|---|---|
| ASC | T | Truncates blanks on input to varying length columns. Default - keep the blanks. |
| DEL | COLDEL , | Column Delimiter |
| | CHARDEL " | Character String Delimiter |
| | DECPT . | Decimal Point Character |
| | | Default characters shown. Valid characters are " % & ' ( ) * , . / : ; < > = ? _ |
| WSF | none | Not used on Import |
| | 1 | Export Lotus 1-2-3/1 or 1A |
| | 2 | Export Lotus Symphony/1.0 |
| | 3L | Export Lotus 1-2-3/2 |
| | 3S | Export Symphony/1.1 |
| IXF | FORCEIN | Ignore codepage differences between IXF file and database on Import. Default - enforce codepage. Not used on Export. |

**Table 12-1** Supported Import/Export File Type Modifiers

- CREATE means that the table is to be created from the import data and then the data added. This option can only be used with IXF format data file. An error is returned if a table already exists with the same name.
- REPLACE_CREATE does the same as CREATE except if the table already exists, its contents are replaced by that from the data file.

The **dcoldata** structure describes the columns to be imported from the OS/2 data file. Use this parameter only if absolutely necessary. Applications usually provide a convenient method of choosing columns for export, in which case you can just import all exported columns. Set **dcoldata** to a null value and Database Services chooses the columns in the natural order from the OS/2 data file.

If the import data format is ASC (non-delimited ASCII) you must specify **dcoldata** in order to describe the column positions of the ASCII data (necessary since data is *not* delimited). Use the **sqldcol** structure (in **sqlutil.h**) to define **dcoldata**.

```
struct sqldcol
{
  short dcolmeth;          /* column specification method   */
  short dcolnum;           /* number of columns specified   */
  struct sqldcoln         /* column descriptors . . . . .   */
  {
    short dcolnlen;           /* length or position . . .   */
    char *dcolnptr;           /* pointer                    */
  }dcolname[1]
};                            /* contents depends on method */
```

For ASC data, the specification method is 'L' for Location (other formats might use Name or Position). For Location, there is only one ordered pair of column descriptors. This ordered pair contains a pointer (in **dcolnptr**) to a table of location pairs. These location pairs describe the start and ending column positions for each column to be retrieved from the data file.

This location pair table can be allocated with the **sqllocpair** structure (defined in **sqlutil.h**).

```
struct sqllocpair
{
  short begin_loc;
  short end_loc;
} locpair[1];
```

The following program fragment gives an idea of how to use these structures to build an import file description for non-delimited ASCII files:

```
struct sqldcol    *dcoldata;      /* ptr to dcoldata          */
struct sqllocpair mypair[2];      /* column start/end pairs   */

dcoldata = (struct sqldcol)malloc(5);     /* allocate space   */
dcoldata->dcolmeth = 'L';                 /* location method  */
dcoldata->dcolnum = 2;                    /* 2 columns        */
dcoldata->dcolname[1].dcolnptr = mypair   /* pointer to table
                                                             */

mypair[1].begin_loc = 1;
mypair[1].end_loc = 6                  /* column 1 - 6 bytes  */
mypair[2].begin_loc = 7;
mypair[1].end_loc = 10;                /* column 2 - 4 bytes  */
```

### *Columns for Export*

For export, the **tcolstrg** structure contains a valid SQL statement. Place the statement in a Database Services varying length character string (**sqlchar** structure).

The **dcoldata** parameter should be left null, unless you wish to change the names of the columns in the exported OS/2 data file. If left null, the output columns have the same name as the database table columns.

To change the column names, set **dcolmeth** to N and place pointers to the new column names in the **dcolnptr** fields of the **sqldcol** structure.

## ENVIRONMENT FUNCTIONS

The functions described in this section are used less frequently than the other Environment routines and SQL statements described in previous chapters. The functions performed by these routines are all available from the Query Manager.

Even though the functions described here are easily accessible through the Query Manager, you may wish to use them in your program if you need to create a database for your application's data rather than keeping that data in a database managed by the user. Chapter 10, "Designing Your Database Program," provides information on installing and maintaining your data within a database managed by the user.

## Creating Databases

Databases are created with the Create Database function call **sqlecred.** Appendix B, "Sample Programs," contains the listing of a C program, CREATEDB.SQC, that uses this routine to create a database requested from the command line. Let's take a closer look at CREATEDB.SQC.

### *CREATEDB.SQC*

You must start the Database Manager before CREATEDB can be executed. Executing the STARTDBM command from the OS/2 command prompt accomplishes this. CREATEDB is very easy to run. Here is an example of how it works:

```
[C:\]CREATEDB d mysample
Enter database password; (press 'Enter' for none): mypw
Enter database comment: My Sample Database

Done!
[C:\]
```

Database "mysample" is created on the D: file system with a password of "mypw."

CREATEDB.SQC declares some global data elements:

```
EXEC SQL INCLUDE SQLCA;          /* SQL Communications area  */
unsigned char password[9] = "\0\0\0\0\0\0\0\0\0";
unsigned char comment[31] = "";
```

The SQLCA **sqlca** is used on all the calls to Database Services. Variables are also declared for the database name and comment. These variables are used on the Create Database function call.

After performing checks on the input parameters, CREATEDB.SQC then creates the database with the **sqlecred** function call.

```
printf("Enter database password; (press 'Enter' for none): ");
gets(password);

printf("Enter database comment: ");
gets(comment);

rc = sqlecred(argv[2],*argv[1],password,comment,0,&mysqlca);
```

The name of the database to be created was passed as the second item on the command line. C places a pointer to this parameter in argv[2]. This value is exactly what is needed for the first parameter to **sqlecred.** The name of the drive on which the database is to be created was passed as the first item on the command line. C places a pointer to this parameter in argv[1]. However, we need to pass the actual character to Database Services, not a pointer to it. So we use *argv[1] to pass the first character of the first parameter. Note that no matter how long the first parameter is, CREATEDB only uses the first character; all others are ignored.

CREATEDB.SQC uses the C **gets** function to obtain character strings from the console.

This is the method used to obtain the optional information from the user (password and database comment). Pointers to these strings are passed on the **sqlecred** function call.

The fifth parameter given to **sqlecred** (0 in the sample program) represents the codepage to be used for the database comment. A zero indicates that the comment codepage should be taken from the codepage active at the time the call is issued.

The last parameter passed to **sqlecred** is a pointer to the SQLCA. Database Services places return code information in **mysqlca.**

***Hints***

When you create a database, you automatically create the entries in the system and volume directories for the new database. The system directory is located on the volume that contains the Database Manager code and the volume directory is on each volume that contains a database. See "Volume and System Directories" in Chapter 6 for details on the Database Services database directories.

The Create Database function call provides a method for forcing the database to be created on the same drive that contains the Database Manager code. The database is created there if the drive parameter is null.

Besides the database name, the system directory also contains an *alias*. This alias is initially set to the same value to which the database name was first set. This alias is really the local name for the database, and you use it on all function calls except for Create Database and Catalog Database. The Database Manager uses the system directory to translate this local name into the original database name. Part of the physical database name is the drive on which it was created. The Database Manager uses the drive information to locate the volume directory for the requested database. Fortunately, your program is isolated from the physical location of the database.

Unlike database names, each database alias in the system catalog must be unique. Thus, if someone provides a diskette containing a database with the same name as one already existing on your system, you need only provide a directory entry with a unique alias for this database.

To rename a database, you must create a new alias for the original database. See "Changing the Database Directory" in this chapter for information on creating an alias for an existing database.

CREATEDB.SQC is an unsophisticated program. Some enhancements that could be made include hiding the password when it is typed in and prompting for its input a second time to let you prove to yourself that you really know what you typed.

## Erasing Databases

Now that you know how to create a database and change two external attributes, password and comment, let's discuss how to erase or *drop* a database. The environment routine that erases a database, **sqledrpd,** is called Drop Database.

***ERASEDB.SQC***

A sample execution of ERASEDB.SQC looks like this:

```
[C:\]ERASEDB mysample
Enter database password; (press 'Enter' for none): mynewpw

Done!
[C:\]
```

All of the database objects are erased, the database directory is removed, the database's entry is removed from the appropriate volume directory, and the system directory entry with the alias used on the Drop Database is removed. Only this one entry is removed from the system directory. If you created any additional entries (with other aliases), those entries remain.

Here is the code that ERASEDB.SQC uses to drop the requested database:

```
printf("Enter database password; (press 'Enter' for none): ");
gets(password);

rc = sqledrpd(argv[1],password,&mysqlca);
```

The only parameters necessary to erase a database are a pointer to an ASCIIZ string containing the database name, (argv[1]), and the current database password.

***Hints***

Drop Database removes only one entry from the system directory. That entry is the one with the database alias equal to the name passed on the Drop Database function call. This allows you to have multiple entries in the system directory for the same physical database name, but with a unique alias. Let's look at an example where you might have such a situation.

Two employees bring you a status database on a diskette. Each employee has named the database STATUS. Since you want a unique name for each database, you create two system directory entries for these databases, each with a unique alias. Figure 12-1 shows the resulting system directory after you create a unique entry for each employee's STATUS database. If you want to remove only one of the employees' directory entries, you uncatalog the appropriate alias name, STATUS1.

*Note:* In the case depicted in Figure 12-1, you cannot drop STATUS because no local entry has that name.

As databases are created, they are assigned new "token ids" to be used in the creation of the OS/2 file system directory. These ids are kept in the volume directory. If a data base is dropped, its token is reused when the next database is created.

## Displaying the Database Directory

If you are building an application that operates on multiple databases, you may want to display a list of existing databases from which your user selects one to work with. The Query Manager presents such a list in the first menu that appears, the Query Manager

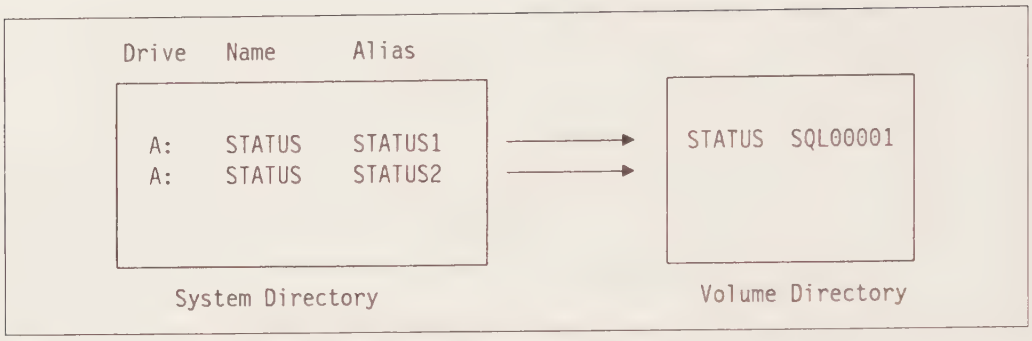



**Figure 12-1** Multiple Aliases for a Physical Database

Databases Main Menu. DBLIST.SQC uses three of the Environment function calls to list the available databases:

- sqledops—Opens a directory scan
- sqledgne—Gets a directory entry
- sqledcls—Closes a directory scan

### *DBLIST.SQC*

The following sample execution was run after executing the IBM sample program SQLSAMPL (which creates the database SAMPLE) and following all the examples shown so far in this chapter (except for the example of erasing a database). The following example lists the contents of the system directory:

```
[C:\]DBLIST
 Dr  Name   Alias  Comment                          Directory  Type  Node
 ──  ─────  ─────  ──────────────────────────────   ─────────  ────  ────
  C: SAMPLE SAMPLE                                            IND
  D: MYSAMPLE MYSAMPLE Still My Sample Database               IND
[C:\]
```

You can also use DBLIST to list the contents of a volume directory. The following example shows how to list the contents of the volume directory on the D drive:

```
[C:\]DBLIST D
 Dr  Name   Alias  Comment                          Directory  Type  Node
 ──  ─────  ─────  ──────────────────────────────   ─────────  ────  ────
  D: MYSAMPLE MYSAMPLE My Sample Database          SQL00001   HOME
[C:\]
```

*Note:* Remember that we used COMMNTDB to change MYSAMPLE's comment in the system directory and left the original comment in the volume directory.

The following program fragment shows the main portion of DBLIST.SQC, the part that retrieves the directory entries.

```
sqledops(drive,&handle,&entries,&mysqlca);   /*open directory*/

for (i=1; i<=entries; i++)
    sqledgne(handle,&myinfo,&mysqlca);/* get directory entry*/

sqledcls(handle,&mysqlca);           /* close directory scan  */
```

Look at the listing of DBLIST.SQC in Appendix B, "Sample Programs," to see how the display formatting is done. Reading a database directory is similar to reading an OS/2 file. You open the directory first, read from it, and close the directory when you are done.

When opening a directory scan, you must decide which directory to scan, the system directory or one of the volume directories. DBLIST.SQC accepts a parameter from the command line designating which directory. If you leave this parameter blank, DBLIST.SQC scans the system directory (the blank is converted by DBLIST.SQC into a X'30' which tells Database Services that you want to scan the system directory). You could also type a 0 on the command line with the same effect.

*Note:* Accepting a drive of 0 to indicate the system directory is different from the other Environment routines that accept a drive parameter. In all other cases (Create Database, Change Database Comment), a null value indicates the system directory.

Opening a scan returns two values to your program, a handle and the number of entries in the directory just opened. Both of these values are used by DBLIST.SQC in the processing of the directory entries. The number of entries is used to control the number of times that the Get Next Directory Entry routine (**sqledgne**) is called.

*Note:* An alternate way of determining when the number of directory entries is exhausted is when the **sqledgne** routine returns a positive number as a return code.

The Get Next Directory Entry routine uses the handle returned on the open call in order to know which scan to process (you can have up to eight scans open in a single OS/2 process). Before we describe the other parameter, &myinfo, you need to understand what Database Services does when you request the next directory entry. Database Services allocates a block of storage to contain the directory entry that is retrieved (you do not do this yourself). This block of storage contains a "snapshot" of the directory entries at the time the function call is issued. Any changes to the database directory after the directory scan is opened are not returned.

Your program must allocate a variable to contain the address of this storage for the snapshot. IBM provides a mapping of this structure, **sqledinfo**, in the sqlenv.h file. The pointer may be allocated with the following statement:

```
struct sqledinfo *myinfo;
```

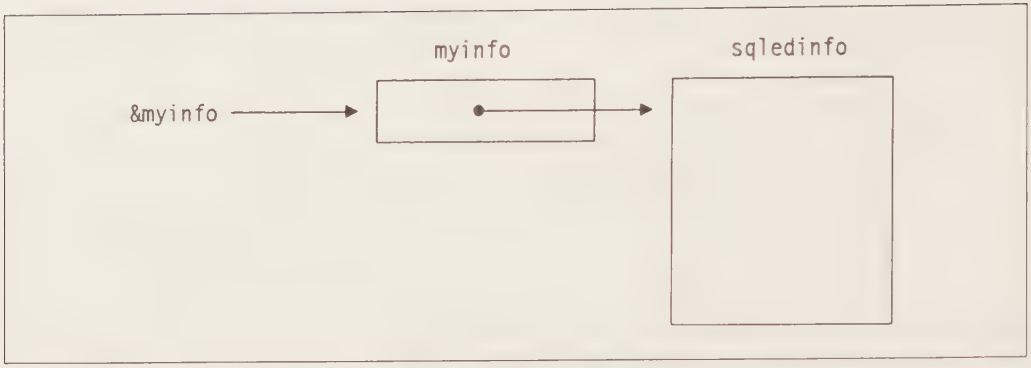



**Figure 12-2** Pointer Relationships in DBLIST.SQC

Upon calling the **sqledgne** routine, you must pass a pointer to your local storage (that is, a pointer to your pointer). This is why DBLIST.SQC passes &myinfo to **sqledgne.** Figure 12-2 shows the relationship between all of these pointers.

DBLIST.SQC formats the output line from the single directory entry contained in the **sqledinfo** structure. The sample program proceeds in this manner until all directory entries have been retrieved. After all entries have been processed, DBLIST.SQC closes the directory scan with the **sqledcls** routine. This routine uses the handle returned on the Open Directory Scan to know which resources (which **sqledinfo** structure) to free.

### *Hints*

Using the information returned by the **sqledgne** routine can be a bit tricky. The character information in the **sqledinfo** structure is not in ASCIIZ form, which is why DBLIST.SQC must copy the information from the returned structure into a local buffer before it can be printed out. The following example should help you understand.

You might think you could print out the returned database name with the following statement, using a pointer to the character string within **sqledinfo:**

```
printf("%s\n",myinfo->dbname);
```

This example does not work. This statement prints out the database name and keeps printing memory contents until a random zero is found. This is because the **sqledinfo** structure has only 8 bytes allocated for the database name (maximum name size), not the nine required to hold the terminating zero. In order to print just the database name, you must copy the 8 bytes from the **sqledinfo** structure into a 9-character array in your program. You can then use the **printf** function to print out the database name.

RENAMEDB.SQC shows the programming required when using a returned directory entry as input to another function call (in this case, the Catalog Database routine). Since the Catalog Database routine expects pointers to ASCIIZ character strings, the directory entry elements must be copied to local storage to make them proper ASCIIZ strings.

## Changing the Database Directory

Every time you create a database on your machine, the Database Manager automatically adds an entry in the system directory for the new database. What if you want to use a database that you did not create? Suppose someone brought you a database on a diskette or you want to use a database on a virtual drive. In neither situation do you have an entry in your system directory for the requested database. Database Services provides a function call, **sqlecatd**, to allow you to create entries in the system directory.

A corresponding function call, **sqleuncd**, allows you to remove entries from the system catalog. Appendix B, "Sample Programs," contains listings for two programs, CATALGDB.SQC and UNCATDB.SQC, that allow you to create and remove system directory entries from the command line. Let's see how they work.

### *CATALGDB.SQC*

Suppose one of your enthusiastic co-workers who also has this book learns how to create a database on a diskette and brings the diskette to you to share in the excitement. Since you have no way to access the diskette database, you use CATALGDB.SQC to create a system directory entry for the new database.

```
[C:\]CATALGDB A mysample
Enter database alias; (press 'Enter' for default): notmydb
Enter database comment: Not My Database

Done!
[C:\]
```

On the command line, you passed the location of the database (A: drive) and the physical name of the database ("mysample"—remember, your co-worker has also been reading this book). If the alias had been left blank, it would have defaulted to "mysample"; "mysample" would be rejected since you already created a local name of "mysample." Thus, you wisely choose to supply a different local name of "notmydb." The supplied comment is placed in the system directory.

At the heart of CATALGDB.SQC is the following program fragment:

```
printf("Enter database alias; (press 'Enter' for default): ");
gets(alias);

printf("Enter database comment: ");
gets(comment);

rc = sqlecatd(argv[2],alias,type,nodename,*argv[1],
comment,0,&mysqlca);
```

After obtaining the optional database alias and comment from the user, a call is placed to **sqlecatd**. This routine requires some of the same parameters as the Create Database call (the database name, the drive on which the database exists, and the database comment and codepage). Let's look at the additional parameters.

An alias may be supplied, but if the alias points to a null string Database Services defaults the alias to the database name (this happens on Create Database because you cannot supply an alias).

### *UNCATDB.SQC*

Once cataloged, you can perform any required operations on your co-worker's database. After returning the diskette, you can use UNCATDB.SQC to remove the system directory entry.

The following command removes the system directory entry for the alias "notmydb," which pointed to the database "mysample" on the A: drive.

```
[C:\]UNCATDB notmydb

Done!
[C:\]
```

Removing an entry from the system directory is very simple. The following code shows the **sqleuncd** function call from UNCATDB.SQC:

```
rc = sqleuncd(argv[1],&mysqlca);
```

The only parameters passed are a pointer to the database alias to be removed and a pointer to the SQLCA.

### *Hints*

Remember that the Catalog Database and Uncatalog Database routines affect only the system directory and no modifications are made to any volume directories. Each successful call to Catalog Database creates a new entry in the system directory.

Of more practical use would be multiple entries for the same database cataloged on multiple devices. Since Database Services only looks on the A drive for a database created on that drive, you might want to create a system directory entry for each possible drive that may contain that database (both A *and* B drives). You can then insert the diskette containing the removable database into either drive.

Uncatalog Database removes a system directory entry but does not erase the database. However, a system directory entry must exist for the physical database in order to use that database. You can use the Catalog Database and Uncatalog Database routines in conjunction to rename a database (that is, to give a physical database a new alias). RENAMEDB.SQC, listed in Appendix B, "Sample Programs," shows how a rename may be accomplished. For example

```
[C:\]RENAMEDB mysample otherdb

Done!
[C:\]
```

renames "mysample" to "otherdb." While not extremely useful, knowing how to rename a database might gain you a few points in a Database Manager trivia quiz.

| Function | Database Manager Configuration | Database Configuration |
|---|---|---|
| Update configuration | sqlfusys | sqlfeudb |
| Return copy of configuration | sqlfxsys | sqlfxdbc |
| Reset configuration | sqlfrsys | sqlfrdbc |

**Table 12-2** Configuration Function Names

## Configuration

Database Services provides three functions to manipulate configuration information. The function names for each configuration category are listed in Table 12-2.

The individual configuration parameters are discussed in detail in Chapter 5, "Database Configuration Parameters." Each parameter described in that section can be read and set via this API. There are some additional parameters available through this API that can be read but not changed. These parameters and their meanings are listed below.

| NAME | MEANING |
|---|---|
| country | Database country code. This is the OS/2 country code that was in effect at the time the database was created. |
| codepage | Database codepage. This is the codepage that was in effect at the time the database was created. |
| release | The release of the Database Manager used to create the database. |

You should rarely need to change the configuration parameters (either Database Manager or database configuration). The configuration API is discussed here for those of you who want to include a configuration interface as part of your application or to use the "read only" parameters described above.

The configuration API works on the basis that each configuration parameter is represented by a unique *token*. You tell Database Services which parameter you are interested in by passing the appropriate token(s) and a pointer to storage for the actual value.

The sample program LOGFILE.SQC contains examples of how to retrieve and set configuration parameters and also changes a database's logfile size parameters. This program shows how to use the **sqlfupd** structure to pass the token/pointer pairs.

```
struct sqlfupd
{
   unsigned short token;
   char           *ptrvalue;
};

struct sqlfupd logfile_array[3];
```

The parameters on the configuration calls (both return and update) include a pointer to the array of token/pointer values (fourth parameter) and a count of how many pairs are being passed (third parameter).

```
sqlfxdbc(database,password,3,logfile_array,&sqlca);
```

If you use the **sqlupd** structure, pointers to the actual values must be cast as (**char** *) because of the declaration of ***ptrvalue.** This can lead to some odd-looking code. The following fragment is taken from LOGFILE.SQC

```
logfile_array[0].token    = SQLF_DBTN_LOGFILE;
logfile_array[0].ptrvalue = (char *)&minlog;
```

where **minlog** is declared as a long integer.

All of the configuration parameters can be represented as integers of the appropriate size. Short integers suffice for most. Some parameters, as the maximum number of log file extensions (maximum value 2,000,000) and the log file size itself (maximum value 2**32-1), require long integers.

The release parameter requires a different data type. The release is actually two parameters (the major release number and the minor release). These two values are stored in a 2-byte field. The high order byte contains the minor release number and the low order byte contains the major release number. Use a pointer to the following structure (not supplied by Database Services) to retrieve the release parameter:

```
struct majorminor
{
  unsigned char minor;
  unsigned char major;
};
```

This is the structure used by LOGFILE.SQC to retrieve the Database Manager release number.

## CONCLUSION

This chapter concludes the discussion on writing database programs. The "do your work" step of the simple database program model presented in Chapter 7 has been expanded to include data manipulation with SQL, creating and deleting databases, and performing utility functions with the Database Services function calls.

We have provided step-by-step instructions on how to use static and dynamic SQL and how to precompile, compile, and link your database programs. Even though SQL simplifies your data management tasks, it brings a new set of considerations to the task of program design. Chapter 10 concentrated on these new considerations: data handling with SQL, managing transactions, and building user interfaces.

Some of the advice given in this part has directed you to supplement your application with elements from the Query Manager. The next part of the book discusses how you work with the Query Manager to create these Query Manager elements.

# PART FOUR

# Supplementing Your Application with the Query Manager

This part of the book is devoted to the Query Manager and how you can use it to provide part or all of a database application. Any Query Manager functions that you can use to support your application should reduce your development cost and implementation efforts.

The Query Manager provides an end-user interface to Database Services, and it includes other functions to find, report, and maintain information stored in a database. An end-user interface is the function provided by a computer program that allows one to use the computer or a computer program without needing to write a program. Stated another way, an end-user interface tells a computer program what you want it to do. In Chapter 1, "What is the Database Manager?," you saw a structural overview of the relationship of the Query Manager to Database Services.

In Part 3, "Writing Database Programs in C," you learned how to build an application using the C programming language and the Database Services programming interfaces. In Part 4, you will discover how Query Manager can supplement your efforts by providing support for all the Database Services utilities and configuration functions. In addition, you may be able to use Query Manager's report writing facility to generate necessary reports that can be browsed or printed.

The information provided in this part of the book is intended to supplement the *IBM OS/2 Extended Edition User's Guide* provided by IBM. This part is most useful to the person who has already installed Query Manager, used it for a few days, and is now ready to learn to use its full power. If you have installed Query Manager, you can try it out by creating a copy of the provided IBM sample database. The sample database is created by using the OS/2 command SQLSAMPL. You can then run the Query Manager and see it in action using a real database.

The first chapter of Part 4, Chapter 13, "Query Manager Usage and Terminology," discusses terminology and how to use the Query Manager. Please read it before you read any of the other chapters. If you are unfamiliar with Database Manager, Chapter 13 will give you an overview of the Query Manager functions.

Chapter 14, "Query Manager Decision Support Tools," discusses the Query and Report tools, which provide a quick and easy way to access and report information in a database. You can use these tools to develop and store predefined queries to supplement your

applications. The stored queries can be exported for distribution and use with your application.

Chapter 15, "Customizing Query Manager," discusses the Procedure, Menu, and Panel tools, which provide a method for defining simple applications for retrieval and maintenance of of database information. These tools can provide a more customized, application-specific user interface to the functions of the Query Manager.

Chapter 16, "Database Operational Support Tools," discusses tools that back up and restore databases and also import and export table data. These tools are easy to use even for the novice.

# CHAPTER THIRTEEN

# Query Manager Usage and Terminology

## OVERVIEW

This chapter provides a brief explanation of the environment in which the Query Manager operates, terms unique to the Query Manager, and the Query Manager's major functions. You need this information to understand the functions discussed in later chapters. The level of information in this chapter assumes that you have minimal knowledge about the Query Manager. Read all of this chapter. Then, based on your interest, read all or some of the remaining chapters in this part of the book.

## QUERY MANAGER OBJECTS

In this part of the book you will learn about the actions you can take with Query Manager. Since these actions are taken on some object, the conceptual user interface provided by the Query Manager is called an Object Action model. In this model, you select an object upon which to act.

A database is an object that contains other objects. These objects were presented in "Database Services Objects" in Chapter 3. You can use the Query Manager to create the following Database Services objects:

- Databases
- Tables
- Indexes
- Views

Query Manager can also modify most of the Database Services objects regardless of whether it or some other application creates them.

The Query Manager has another set of objects. You will recognize these since they parallel the functions provided by Query Manager. The Query Manager objects are

- Queries
- Forms
- Procedures

- Menus
- Panels

You use these objects to store the definitions of the specific Query Manager function to be performed. For example, a Query object stores the definition of how a database query is to be performed (that is, what table, what rows, and so on will be used). The object is stored in the database that you are using at the time it is defined. Query Manager keeps track of its objects and only the Query Manager can manipulate them. The Query Manager objects in one database cannot be used from a different database if you have multiple databases.

To summarize, when you start Query Manager, it presents a list of databases for you to select from. Query Manager then presents you with lists of objects contained within a database that you can act on. The objects are either Database Services objects or Query Manager objects.

## USING THE QUERY MANAGER

### The OS/2 Desktop

Before discussing the Query Manager, we'll define a few terms that describe the OS/2 graphical user interface. When you start OS/2, the system displays a screen called the *desktop*. The *desktop* has a Start Programs window and some icons. Figure 13-1 depicts the desktop.

Please note that this figure does not contain all the detail that you will see but is used to show the major elements of the desktop. The icons are small vertical rectangles that appear at the bottom of the desktop. Windows are larger horizontal rectangles that appear above the icons. The window and icons can be moved around on the desktop and can be changed in size. The icons are small graphical depictions of some task (function) that OS/2 is running.

There are two major windows on the desktop.

1. **Start Programs** This window is shown on the left of the desktop in Figure 13-1. As you select programs from the Start Programs window to be run, an icon is created to represent each program. When you install Query Manager, the **Query Manager** entry is added to your Start Programs list. After Query Manager is started, when you return to the desktop Query Manager is represented by an **OS/2** icon at the bottom of the desktop and as an entry in the Task Manager's list.

2. **Task Manager** This window, shown on the right of the desktop in Figure 13-1, is simply a list of those programs that have been started. For example, if you have configured your system to run the PC/DOS compatibility function, the Task Manager will have an entry for the **DOS Command Prompt.** The Query Manager appears as **Query Manager.**

In summary, the Start Programs window allows you to start applications, and the Task Manager window allows you to switch between applications you have already started. You

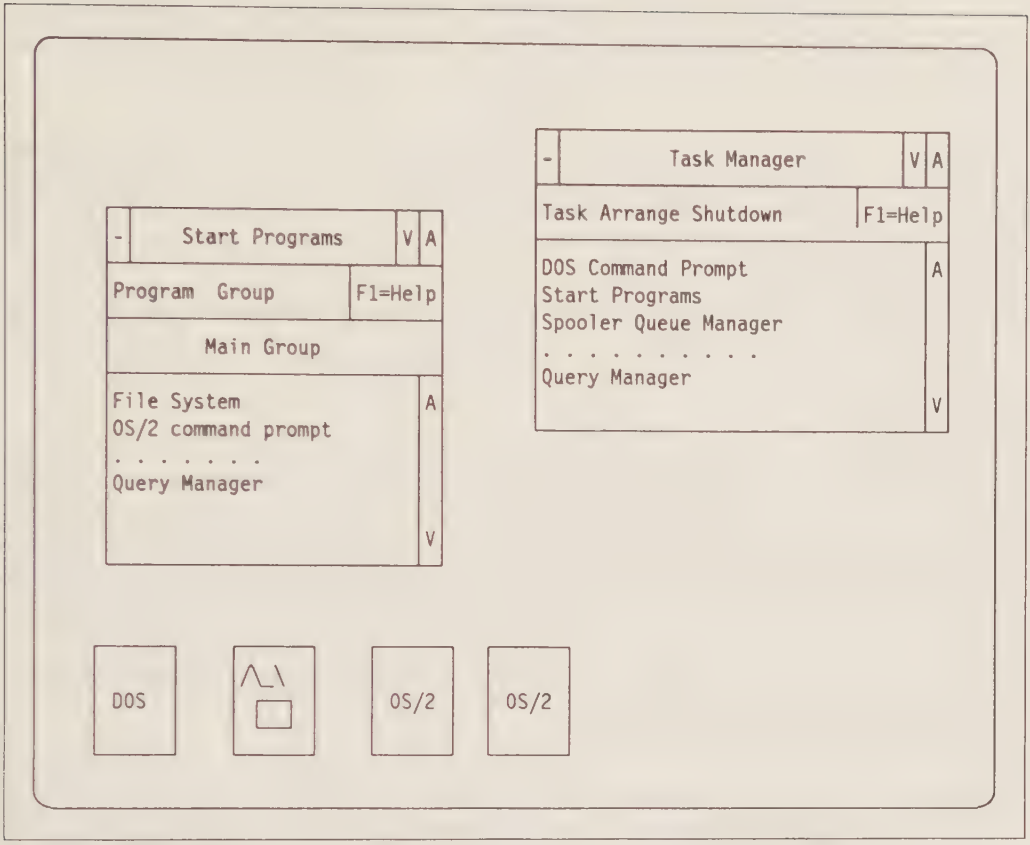



**Figure 13-1** OS/2 Desktop

can always return to the Task Manager window on the desktop by pressing the Ctrl and Esc keys together. Once on the desktop, you can go to either the Start Programs or Task Manager windows.

### Starting the Query Manager

After you install Database Services and the Query Manager, you have two methods for starting the Query Manager from the OS/2 desktop.

1. **In the Start Programs window** Select the **Query Manager** entry and press Enter. This method starts an OS/2 command session, then starts the Query Manager. When you exit the Query Manager, the OS/2 command session is terminated and you are returned to the Start Programs window.

   If you have a mouse pointing device, you can easily move the pointer to the **Query Manager** entry in the Start Programs window and click button one twice. This is equivalent to pressing the Enter key.

2. **From an OS/2 command prompt** If you have already started an OS/2 command prompt, type `QUERYMGR` at the command prompt and press Enter. This method returns you to the OS/2 command prompt when you exit the Query Manager.

Although you start Query Manager from the OS/2 desktop, its menus (windows) are not subwindows of the desktop. The Query Manager operates in its own screen group and acts as if it owns the entire screen display area. Operating in this mode prevents you from changing the size of Query Manager's windows or from exchanging data from its menus with another application.

Customizing the startup process and other suggestions are described later in "Using OS/2 Commands" in Chapter 14.

### Screen Format

This information is provided for those readers who have not used the Query Manager or are just starting to use it. Definitions are provided for Query Manager terms used throughout the rest of this book. This information may also help you get an idea of the "look and feel" of Query Manager. This section is especially important if you are going to use Query Manager to supplement your application. It will be much easier for your application user to switch between your application and Query Manager if the interaction techniques are similar.

The first screen displayed after you start Query Manager (Query Manager Databases Main Menu) is shown in Figure 13-2.

The screen has three distinct areas:

**Action Bar** The top line is the action bar area, which contains the actions that can be taken. These actions are represented by single words. For example, Figure 13-2 lists these possible actions:

- Actions
- Tools
- System
- Exit
- Help

Help is always available by pressing the F1 key. It provides context-sensitive information about actions, entry fields on menus, and provides further explanation and possible recovery actions when error messages are displayed. The other actions vary depending on the menu.

**Title Line** The second line holds the title of a menu and the scrolling indicators (scroll). *Databases* is the title of the above menu. The scrolling indicators indicate whether there is more information than can be displayed. Scrolling can be done left and right and up and down.

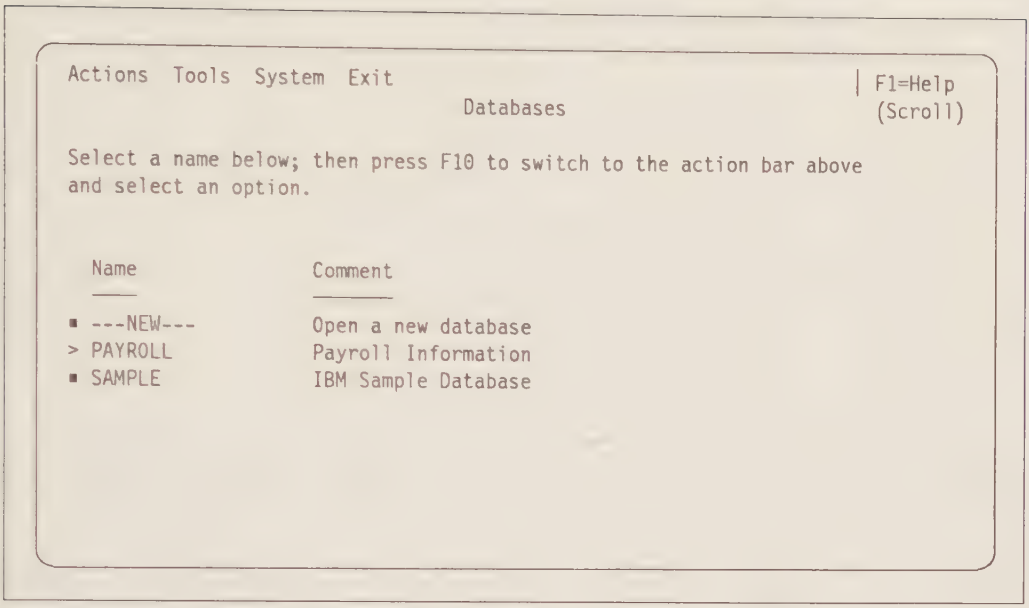



**Figure 13-2** Query Manager Databases Main Menu

**Body Area** The screen area below the title line.

***Menu Types***

The Query Manager uses several types of screens.

**Main Menus** Used to select an object on which to act. Main menus occupy the entire screen. They are distinguished from other menus in that they have no border around them. Figure 13-2 is a main menu.

**Pop-up Menus** Used to display status information or gather information to perform a requested function. Pop-ups overlay either a main menu or another pop-up. They appear in the body area and are always framed. Figure 13-3 depicts a pop-up over a main menu.

**Pull-down Menus** Allow you to select the action to be taken on an object. Pull-downs are framed menus whose upper edge touch the action bar. They appear when an action in the action bar is selected. Figure 13-4 depicts a pull-down from the action bar of a main menu.

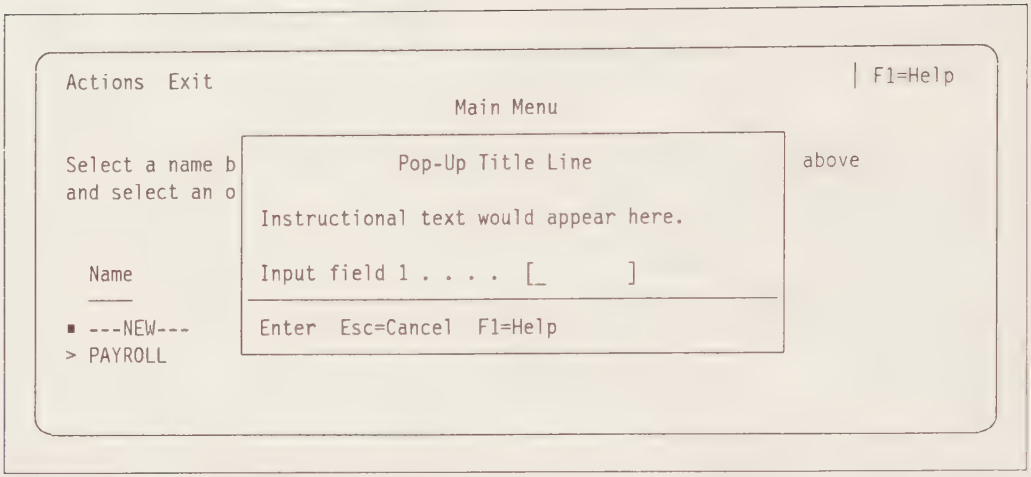



**Figure 13-3** Example Pop-up Menu

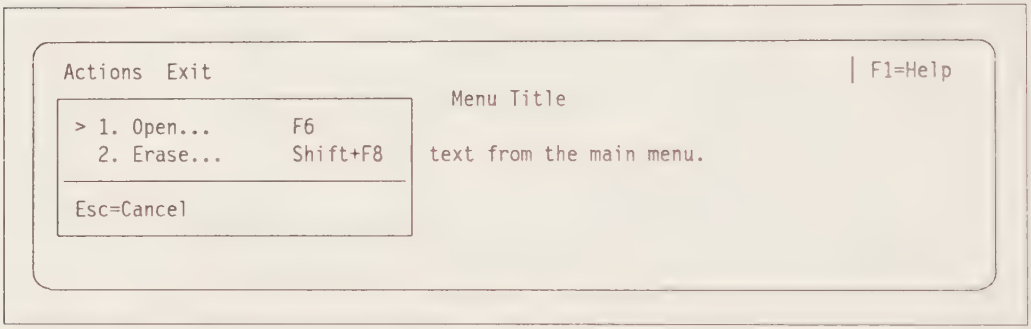



**Figure 13-4** Example Pull-down Menu

### *Selection Techniques*

There are many ways to select an item in an object list within a menu. You must first move the cursor to the object to be selected. This can be done with either the arrow keys, the tab keys, or a mouse. Press the Enter key to select the object. The space bar can also be used in many cases.

The F10 key is a special key in this process. After you select an object, you are then ready to pick an action to take on the object. F10 is the key that switches you between the Body Area (where you select objects) to the Action Bar (where you pick actions).

Several other keys can cause movement between screen areas or the screen contents to be changed. Using these keys and the mouse are documented in detail in the *IBM OS/2 Extended Edition User's Guide*.

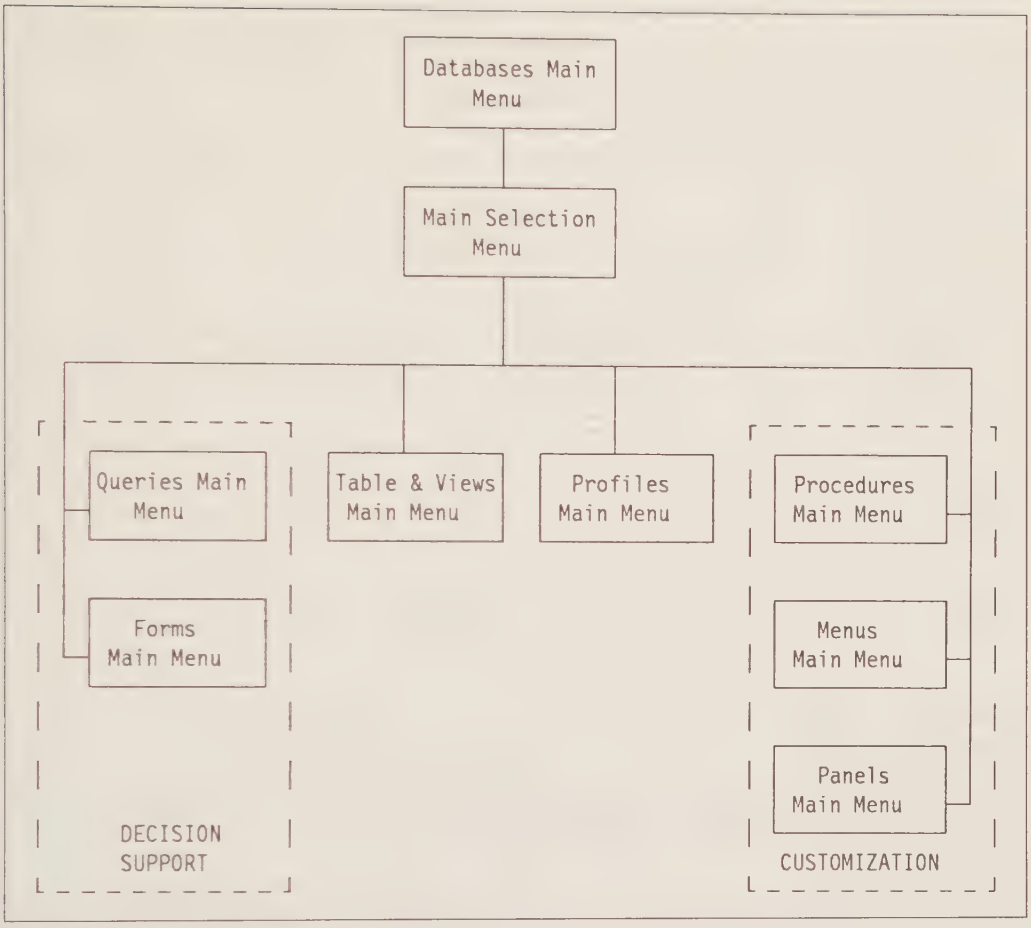



**Figure 13-5** Query Manager Menu Roadmap

## Screen Navigation

The Query Manager consists of a set of hierarchically related menus that lead you through a dialog of choosing objects on which to take actions. This hierarchy is sometimes called an inverted tree in which you start from a "root" menu and proceed out to the branches of the tree. Subdialogs (or subtrees of the hierarchy) are provided to collect information needed to perform an action on an object.

Figure 13-5 provides a menu tree of the major Query Manager menus. Each bottom level menu has sub-menus and pop-ups associated with them.

The dashed boxes in this figure show the menus used to address two of the three major end-user tasks that can be done using the Query Manager. The other end-user task, Operations Support, is done using tools associated with several main menus.

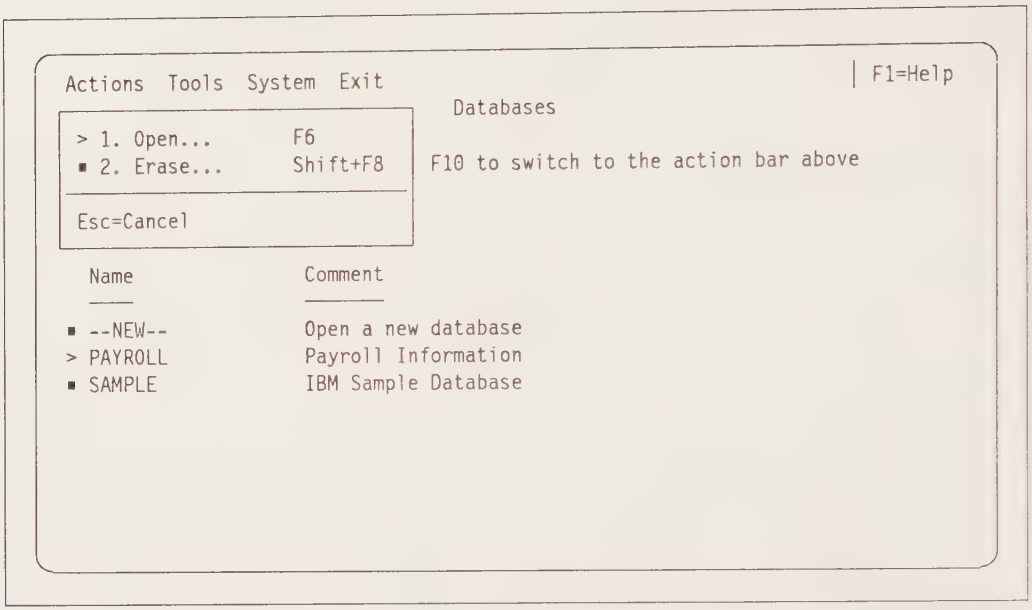



**Figure 13-6** Query Manager Databases Main Menu, Actions Pull-down

### *Databases Main Menu*

Now that you've had some explanation of the major areas of a screen and the types of screens, let's return to the Query Manager Databases Main Menu to learn more about its purpose. The Query Manager Databases Main Menu allows you to select a database to use with the Query Manager. In this menu, as with most of the menus, you select an object (a database) on which to perform an *action*. After selecting the database, an action can be selected from the action bar. Selecting an action causes a pull-down containing *options* to appear. Figure 13-6 shows the list of actions can that can be performed on databases.

The option most usually taken is to **Open** an existing database. This option can also be used to create a new database by selecting the ---***NEW***--- object. The **Erase** option removes an existing database.

Once you have selected a database, you usually **Open** it so that you can use the other Query Manager functions to manipulate the objects contained in that database. Figure 13-6 shows the screen after you have selected a database, switched to the action bar (F10), and selected the **Actions** pull-down so that you can **Open** the database.

You can use some shortcuts to avoid performing common options while in the Query Manager. For example, if you select the database then press the F6 or Enter key to **Open**, you can skip pressing F10 and Enter. Use the arrow keys to move the cursor to the option or object to be selected. Mnemonics and numbered keys provided for most lists enable you to skip moving the cursor and pressing Enter to make the selection. As you become more familiar with Query Manager, you will learn these and other shortcuts.

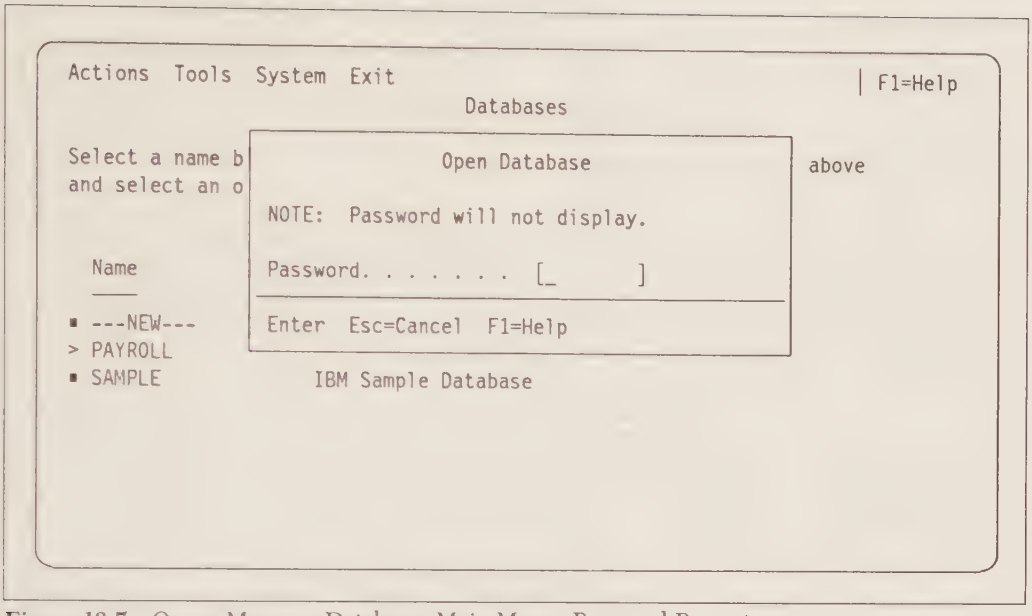



**Figure 13-7** Query Manager Databases Main Menu, Password Prompt

After selecting the **Open** option as shown in Figure 13-6, Query Manager displays the screen shown in Figure 13-7 to obtain the password for the database. This is a pop-up menu. Remember that a pop-up looks like a box and overlays a main menu.

At this point, type the database password and press Enter. If the database has no password, simply press Enter.

### *Main Selection Menu*

Next you see the Main Selection Menu (Figure 13-8). The Main Selection Menu is the navigation hub that is used to move from one functional area of the Query Manager to another. The name of the database in use appears in the title line.

The following list explains the Main Selection Menu's functions.

1. **Tables and Views** You use this function to run many of the tools and utilities provided by Database Services that create and erase tables, views, or indexes. You can choose to add or modify data in tables. Some of these functions are discussed in Chapter 16, "Database Operational Support Tools."

2. **Queries** The Queries function is the major function that supports the "decision maker" and is used to query one or more existing tables. You need not be familiar with the SQL language to construct a query using this function.

3. **Forms** After running a query, the results are displayed in a report format. You can modify the report format directly using the Forms function from the Queries function.

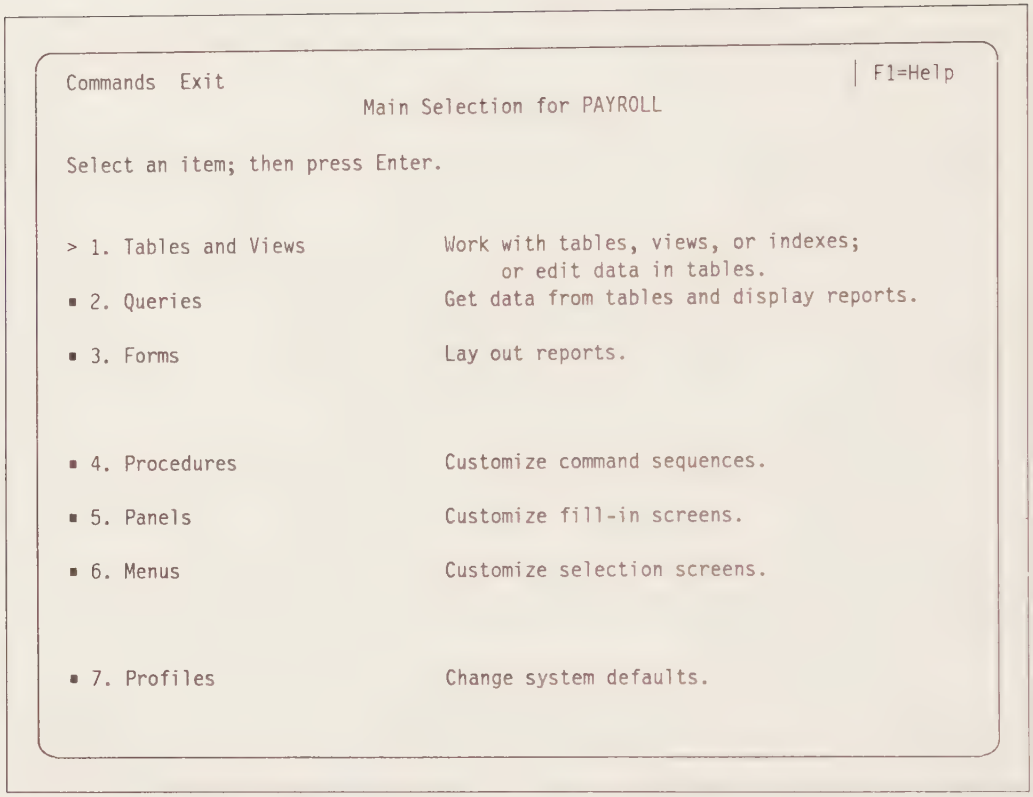



**Figure 13-8** Main Selection Menu

These functions are discussed in greater detail in Chapter 14, "Query Manager Decision Support Tools."

4. **Procedures, Panels, and Menus** These functions implement customized applications and are discussed in Chapter 15, "Customizing Query Manager."

5. **Profiles** You can customize your Query Manager options using this function. It is explained in "The Query Manager Profile" in Chapter 14.

## Query Manager Terminology

The Query Manager uses different terms than SQL because it assumes the user is unfamiliar with SQL terminology. This may be confusing for those familiar with SQL. Table 13-1 shows an equivalence of Query Manager terms used to describe data types and attributes with the SQL terms.

| Query Manager Term | SQL Term |
|---|---|
| Character (fixed length) | CHAR |
| Character (variable length) | VARCHAR |
| Small integer | SMALLINT |
| Large integer | INTEGER |
| Decimal | DECIMAL |
| Special data | LONG VARCHAR |
| System date and time | TIMESTAMP |
| Scientific notation | FLOAT |
| Time | TIME |
| Date | DATE |
| Text (no) | FOR BIT DATA |
| Data required | NOT NULL |

**Table 13-1** Query Manager versus SQL Terms

## CONCLUSION

In this chapter you learned about Query Manager usage and terminology. As you can see, the name Query Manager understates the functions offered. In addition to queries, the Query Manager lets you use many of the Database Services functions without doing programming. By maximizing these functions, you can provide applications more quickly and at a lower cost.

The remaining chapters of this book investigate the functions of the Query Manager in greater detail.

# CHAPTER FOURTEEN

# Query Manager Decision Support Tools

## OVERVIEW

One of the advantages of collecting data on a computer is that you can look at data in many ways and change it. A database system makes the job easier by letting you extract just the data you want, then presenting that data in a sequence that helps you to better answer questions. Decision support tools help you perform these tasks.

Query Manager's Prompted Query is a decision support tool. One of the Query Manager's best functions allows a user without knowledge of the SQL language to retrieve data from tables and then present it in a tabular report format. In this chapter, you will learn about Prompted Query and the direct SQL Query function that you can use in two general ways.

1. **Develop predefined queries and reports** You can use the query and report writing capability to supplement your application instead of writing programs. The query and report definitions can be exported and distributed with your application, then installed with the rest of your application. The end user can further tailor these queries and reports to meet their needs by using the Query Manager.

2. **An ad hoc query tool** It is difficult to determine ahead of time the information someone will need. Understanding the query and reporting capabilities will help you determine whether Query Manager can be used as an ad hoc query tool by your application user. This information is covered in "The Ad Hoc Query Process" below. If useful, you could include documentation on the names of the SQL tables and views created by your application, the column information the tables contain, and how the tables are related. Using Query Manager may meet the end user's needs and provide a much greater degree of flexibility.

We assume that you have read the *IBM OS/2 Extended Edition User's Guide* section on using the Database Manager or that you have installed the Query Manager and are already somewhat familiar with it. The IBM-provided documentation contains introductory information about the query and report writing tools. This chapter is intended to supplement the IBM documentation and highlight information that may not be obvious to a beginning user. The first few sections of this chapter will be high-level information followed by very detailed information not contained in the IBM documentation.

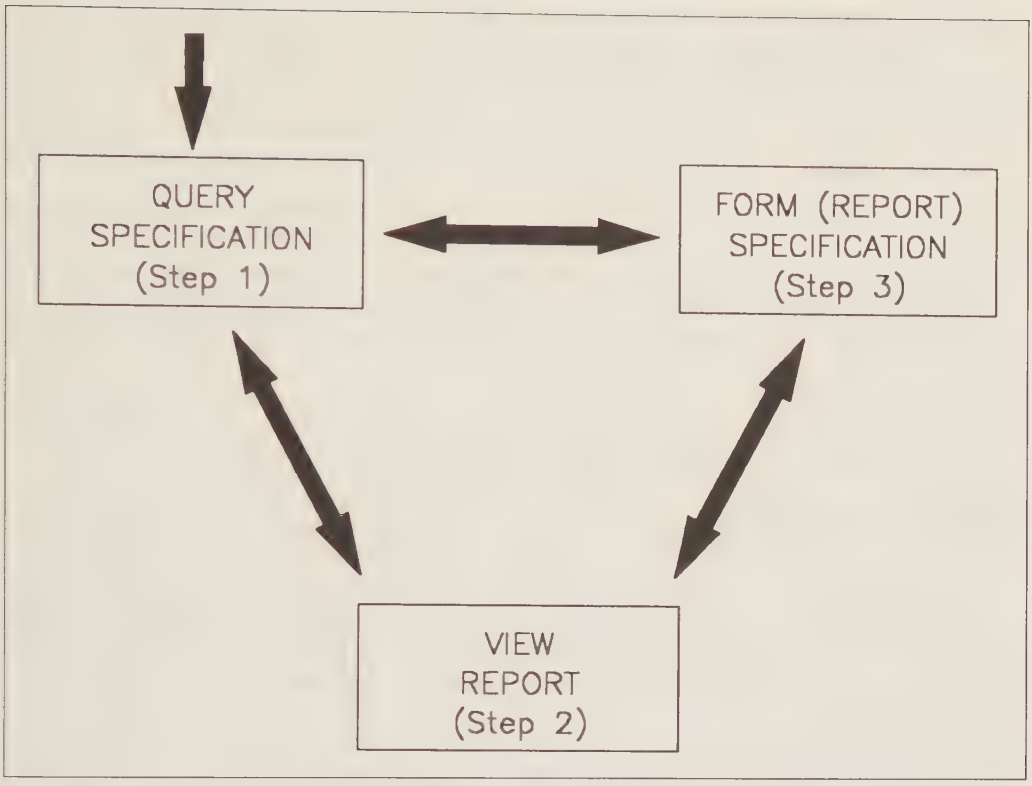



**Figure 14-1** Query–Form–Reports Triangle

## THE AD HOC QUERY PROCESS

Figure 14-1 provides an overview of the ad hoc query process.

The decision process usually begins with a problem or question. To find answers to your problem, you usually make some sort of hypothesis about the causes. To verify your hypothesis or to get background information about the problem, you may want to look at the data stored in your database tables. This is called the *query process*.

Let's use an example to demonstrate the query process. Suppose you are having unexpected slow sales of Fluffy Soap in the Western Sales region and want to find out why. The query process as presented in Figure 14-1 consists of the following steps:

1. **Formulate the query.**

   The first step in formulating a query (query specification) is to determine what table(s) contain pertinent information. In our example, this includes tables containing sales volumes, product codes, sales office name, and location. In addition, you should consider

if the data is to be summarized or needs to be ordered to help in your analysis. If so, you should specify sorting of the query results. Finally, since you do not want all the data, you need to specify query selection criteria to include only Fluffy Soap products and sales from the Western Region.

2. **View the report.**

   The results appear in a report after you formulate the query and run it. These results may or may not answer the questions you have. If you decide your query was not exactly right, you can return to step 1 and refine your query. For example, suppose you specified the sales region code incorrectly and recognized that you were looking at Eastern Region sales data. You would then want to return to step 1 and correct the sales region code requested.

   If your questions are answered, you can quit. Query Manager gives you the option of saving your query and report specification for later use. If they are saved, you can later export them to an ASCII file for inclusion with your application.

3. **Alter the report format.**

   Perhaps the report contains the information you need, but it is not presented very effectively. To make minor changes in how the same data is presented, you can alter the report format and return to continue reviewing the report. In our example, you will need to modify the report format to get the data summarized or you may have three different sizes of Fluffy Soap and want to see the sales figures totaled or averaged by city in the region.

   A default report format is provided based on the data type and attributes of the columns selected in the query. If you retrieve several fields from a table, they probably will not all fit on the display screen at once. Although you can scroll the view port over to the right to see the additional fields, it will be difficult to relate two fields that are on different screens. You can alter the report so that the two fields appear side by side, thus allowing you to make visual comparisons of the information.

   After altering the format, you can continue reviewing the information. If you need to change the query while altering the report format, go directly to the query specification step without returning to review the report.

## Starting the Query Process

You start the query process from either a SQL Query or a Prompted Query. A SQL Query is the Query Manager term for a single SQL statement. The following statement is an example you might use to find out all the tables in the database:

```
SELECT CREATOR, NAME FROM SYSIBM.SYSTABLES
       ORDER BY CREATOR, NAME
```

The SQL Query statement can also change or create a database object. This function is

provided for the user who is SQL literate and may wish to take full advantage of the SQL language to do some things that can not be done in other ways with Query Manager.

Some SQL capabilities are not addressed by Prompted Query. If you want to SELECT data that requires the following SQL functions, you must build your own SQL SELECT statement using the SQL Query function.

- UNION, INTERSECT, or EXCEPT
- Summary functions (AVG, SUM, etc.) inside an expression
- Summary functions with DISTINCT
- HAVING clauses
- GROUP BY
- Subquery. This is a SQL SELECT that has as its operand another SELECT statement. You can accomplish this indirectly with Prompted Query by running a query and saving the result in a table. You can then do a Prompted Query against that result table.

You must also use a SQL Query to have variable prompting occur. See "Using Variables" in this chapter for a discussion about variable usage.

Many of the Query Manager menus provide a **Command** action. Go to the action bar and select the **Command** action, then select the **SQL Query** option. Query Manager displays a blank menu in which to type a SQL statement. SQL Queries can also be created by converting a Prompted Query to SQL. After you create queries with the Query Manager, they can then be exported and imported into another database by the Query Manager.

## PROMPTED QUERY

You should use Prompted Query if you are unfamiliar with SQL syntax and the language capabilities. These are some of the questions asked by novice users that can be avoided by using Prompted Query:

- Where does that parenthesis go?
- What are the summary functions?
- What are valid expressions?
- How do I prefix or suffix search a character string?
- What is the correct format for a TIMESTAMP column?
- How do I do a join between two tables?
- How do I sort the result in descending sequence?

It also provides a convenient interface for those of us who do not have perfect memories regarding the following:

- What tables and views are in the database?
- What are the exact names of the tables, views, and columns?
- What columns are in this table?
- What is the data type of a column? You can perform only certain operations on particular data types. Prompted Query allows you to pick only valid operations.

People who constantly use SQL have no trouble stating the information they want and how they want it. However, if you are like most users and want to maximize SQL use without remembering the details, you probably will become a regular user of Prompted Query.

To initiate a query, from the Main Selection Menu, select the **Queries** option. Query Manager displays the Queries Main Menu, which provides a list of all previously saved queries. To create a new query, select the object ***---NEW---***, and then choose the **Open** option of Actions. Selecting an existing query allows you to rerun that query.

Prompted Query constructs a single SQL SELECT statement for you after asking you questions about what information you want from the database. We will not attempt to describe the step-by-step process that Prompted Query takes you through: The best way to learn Prompted Query is to use it.

Figure 14-2 shows the result of a Prompted Query dialog to build a query that finds people earning more than $10,000 in the STAFF table of the IBM sample database. A join is done with the ORG table to obtain the name of the department the person belongs to.

After completing the Prompted Query dialog, you can run the query and see the report. Figure 14-3 is the report output you would see for the Prompted Query shown in Figure 14-2.

You can see the SQL statement generated from your Prompted Query dialog by choosing the **Show SQL** action from in the **Actions** pull-down on the Prompted Query Menu. Figure 14-4 shows the SQL generated for the example in Figure 14-2.

Displaying your SQL Query can be useful for those interested in learning SQL. You can use the companion function **Convert to SQL** to generate the SQL statement and save it for later usage. This may be useful for prototyping the SQL statements being developed for a program you are writing in another language. As discussed previously, Prompted Query has some limitations. However, you can use it to generate a SQL statement that you would subsequently modify to get the desired SQL statement.

### Using Variables

It is sometimes helpful to rerun the same query but change the selection criteria of one or two of the columns. For example, if you have a query that extracts sales data for one sales region, you might want to see exactly the same report for a different region. This can be done using a Query Manager variable as the value in the selection criteria.

```
Actions  Edit  Specify  Display  Exit                                  | F1=Help
                        Prompted Query ---NEW---

Select Edit above to revise a query or select Specify above to add entries

 Tables:
 ▪ BILL.STAFF(A)
 ▪ BILL.ORG(B)

 Join Tables:
 ▪ A.DEPT  And B.DEPTNUMB

 Columns:
 ▪ NAME
 ▪ DEPT
 ▪ SALARY
 ▪ DEPTNAME

 Row Conditions:
 ▪ If SALARY Is Greater Than 10000

 Sort:
 ▪ Ascending by DEPT
 ▪ Ascending by NAME
```

**Figure 14-2** Prompted Query Menu

A Query Manager variable is indicated by prefixing a name with an ampersand (&). A name is a string of characters beginning with an alphabetic character. For example:

```
&REGION
```

When the Query Manager runs a SQL Query, it scans the statement for variables. When encountered, a substitution is made for the variable using the value of the variable from the Query Manager's Variable Pool. If the variable has not been defined (that is, does not exist), the user is prompted by the Query Manager to provide a value in a pop-up menu.

Variables can only be used in SQL queries, not Prompted queries. You can construct your query using Prompted Query and test it to make sure it gets the desired data. After testing the query, you can then use the **Convert to SQL** function to make it a SQL Query. The SQL Query can then be changed to include a variable(s). Remember, prompting only occurs when the variable has not previously been defined. Therefore, if you rerun the query via the query specification menu, you will be prompted only on the first run for the

```
Actions  Display  Exit                                          | F1=Help
                                  Report                  More:     ■↓


NAME         DEPT      SALARY  DEPTNAME
---------  ------  ----------  --------------
Daniels        10    19260.25  Head Office
Jones          10    21224.00  Head Office
Lu             10    20010.00  Head Office
Molinare       10    22959.20  Head Office
Hanes          15    20659.80  New England
Kermisch       15    12258.50  New England
Ngan           15    12508.20  New England
Rothman        15    16502.83  New England
James          20    13504.60  Mid Atlantic
Pernal         20    18171.25  Mid Atlantic
Sanders        20    18357.50  Mid Atlantic
Sneider        20    14252.75  Mid Atlantic
Abrahams       38    12009.75  South Atlantic
Marenghi       38    17506.75  South Atlantic
Naughton       38    12954.75  South Atlantic
O'Brien        38    18006.00  South Atlantic
Quigley        38    16808.30  South Atlantic
```

**Figure 14-3** Sample Report from STAFF

variable's value. (In other words, you must exit from the query and then request that it be rerun from a higher level menu.)

You can also use variables in the Report Form to supply meaningful information. An example would be to put a variable in the Heading Text identifying the sales region name. You could use the same or a different variable from that used in the query.

There are some special variable names with respect to the Report Form. If you prefix a number with an ampersand, the Query Manager assumes this is a variable that refers to the value of the corresponding column in the query. In our example, if Sales Region Name is the first column requested in the query, using an "&1" in the Heading Text causes Sales Region Name to be printed. The SQL Query might look like this:

```
SELECT REGION_NAME, SALES, ITEM_CODE
  FROM PRODUCT_TABLE
    WHERE REGION_NAME = '&REGION'
```

In the Heading Text, you could reference either &REGION or &1 and get the sales region's name since they would be identical. You could also suppress printing the column (using the **OMIT** option) and still use it in the Heading Text.

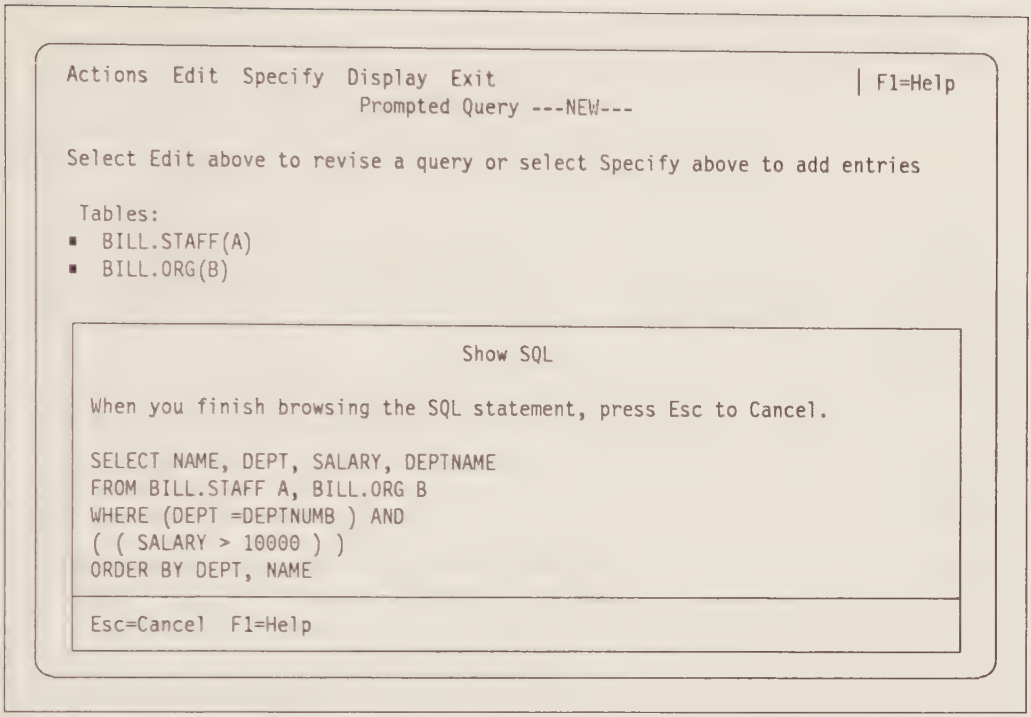

Actions Edit Specify Display Exit
F1=Help
Prompted Query ---NEW---
Select Edit above to revise a query or select Specify above to add entries
Tables:
BILL.STAFF(A)
BILL.ORG(B)
Show SQL
When you finish browsing the SQL statement, press Esc to Cancel.
SELECT NAME, DEPT, SALARY, DEPTNAME
FROM BILL.STAFF A, BILL.ORG B
WHERE (DEPT =DEPTNUMB ) AND
( ( SALARY > 10000 ) )
ORDER BY DEPT, NAME
Esc=Cancel F1=Help

**Figure 14-4** Generated SQL

## Saving Specifications

You can save either the query (Prompted or SQL statements) or form you have specified. In fact, when you exit from either query or forms you are asked whether or not you want the query or form saved.

If you save a query or a form, it is stored as a Query Manager object. There are several things you can do with the query or form:

1. **You can export these objects to an OS/2 file.** This allows you to include it with your application. When your application is installed, these Query Manager objects must be imported into the application's database. See "Packaging Your Queries" in this chapter for more information.
2. **You can reuse the definition again.** You can then simply select the saved query to be rerun from the Query Main Menu. This is handy technique for saving standard queries you run repeatedly.
3. **You can change (edit) the definition at a later time.**
4. **You can use the definition in a customized procedure.** See Chapter 15, "Customizing Query Manager," for further information.

## FORM SPECIFICATION

A report form defines how the results of a query are presented or, in other words, how the report data is formatted. You can use the report form to determine the horizontal order of the data columns presented and to specify how many output positions each column will take, as well as other editing information. These options are covered in detail in the IBM documentation.

A report form can be specified or modified directly from the Main Selection Menu. Select the **Forms** option. Query Manager displays the Forms Main Menu, which provides a list of all previously saved forms. To create a new form, select the object ***---NEW---***, then choose the **Open** option of Actions. Selecting an existing form (that is, one previously created) allows you to edit that form.

The following subtopics cover some subjects that are frequently of interest to novice users and may not be obvious from the IBM documentation. They may help you create a report or avoid usage problems.

### Multiple-column Usages

A report column can only have a single usage. However, in some reports you may want to have multiple uses for the same column. For example, you might want to have the SUM and AVG (average) of SALARY by DEPT in the query we wrote to find those people who make more than $10,000 in the STAFF table (see Figure 14-3).

You can accomplish this by selecting the SALARY column to appear twice in the query. A column can appear as often as you wish. You can have a different usage applied to each column in the report. In this case, you type `BREAKn` on DEPT. Type `SUM` for one of the SALARY columns and type `AVG` for the other SALARY column.

After the report is displayed, select the **Display** action on the action bar. Choose the **Form** action from the pull-down. Query Manager will display a menu that looks like Figure 14-5, but nothing will appear in the input areas below Usage.

Move the cursor to the Usage column and type `BREAK1` for DEPT, `SUM` for SALARY, and `AVERAGE` for SALARY1, as shown in Figure 14-5. Select the **Display** action on the action bar. Choose the **Query** action from the pull-down. Query Manager will display the report with the requested information as shown in Figure 14-6.

### Breaks and ORDER BY

If you use a break, be sure that you have specified the same columns to be sorted by using an ORDER BY clause. The sort order should be done in the same sequence as the break specifications. If not, you will receive no error messages, but the report lines will not appear together.

Two sample reports will be shown which illustrate the correct and incorrect specification of the BREAK and ORDER BY functions. The data table used to produce both reports is shown in Table 14-1.

The correctly specified report has two break levels, DEPT_NUMBER and JOB, respectively. Figure 14-7 shows the correct report where the sort order and break specifications match.

```
Actions  Specify  Display  Exit                                  | F1=Help
                          Form MYREPORT

Num  Column heading                Usage        Indent  Width   Edit     Seq
---  ---------------               -----        ------  -----   ----     ---
  1  [NAME                      >  [         ] [2  ] [9     ] [C     ] [1  ]
  2  [DEPT                      >  [BREAK1   ] [2  ] [6     ] [L     ] [2  ]
  3  [SALARY                    >  [SUM      ] [2  ] [10    ] [L2    ] [3  ]
  4  [DEPTNAME                  >  [         ] [2  ] [14    ] [C     ] [4  ]
  5  [SALARY1                   >  [AVERAGE  ] [2  ] [10    ] [L2    ] [5  ]
```

**Figure 14-5** Report Usage Code Specification

```
Actions  Display  Exit                                         | F1=Help
                                Report                    More:     ■↓

NAME          DEPT      SALARY  DEPTNAME            SALARY1
---------     ------  ----------  --------------   ----------
Daniels         10    19260.25  Head Office         19260.25
Jones                 21224.00  Head Office         21224.00
Lu                    20010.00  Head Office         20010.00
Molinare              22959.20  Head Office         22959.20
                  *  ----------                     ----------
                      83463.45                      20865.86

Hanes           15    20659.80  New England        20659.80
Kermisch              12258.50  New England        12258.50
Ngan                  12508.20  New England        12508.20
Rothman               16502.83  New England        16502.83
                  *  ----------                     ----------
                      61929.33                      15482.33

James           20    13504.60  Mid Atlantic       13504.60
Pernal                18171.25  Mid Atlantic       18171.25
Sanders               18357.50  Mid Atlantic       18357.50
```

**Figure 14-6** Report with Multiple Usage Codes

| NAME | DEPT | JOB | YEARS | SALARY |
|---|---|---|---|---|
| James | 20 | Sales | – | 13504.60 |
| Sanders | 20 | Mgr | 7 | 18357.50 |
| Pernal | 20 | Sales | 8 | 18171.25 |
| Marenghi | 38 | Mgr | 5 | 17506.75 |
| O'Brien | 38 | Sales | 6 | 18006.00 |
| Hanes | 15 | Mgr | 10 | 20659.80 |
| Quigley | 38 | Sales | – | 16808.30 |
| Rothman | 15 | Sales | 7 | 16502.83 |
| Koonitz | 42 | Sales | 6 | 18001.75 |

**Table 14-1** STAFF Table

Suppose that you forget to include the JOB column as the second column to be sorted when you specify the query. As before, you modify your report specification to have both DEPT_NUMBER and JOB as control breaks. Figure 14-8 shows the result of your error.

As you can see, DEPT_NUMBER 20's JOB TOTALs are incorrect. There are no error messages to indicate your oversight. Depending on the physical order of the data, you may get what appears to be a correct report. The error may even go undetected until some time later when the contents of the table are changed and the report is rerun.

If the accuracy of your report is important, take time to review it and run it against different sets of data to test it before using the summary information.

### Multiple Column Heading Lines

You can get up to 8 lines of column heading text. If you want multiple column heading lines displayed for your report, you must **Display** the report and alter the column heading text. Wherever an underscore character is found in a column name, the remainder of the name is placed on the next line. The side effect of this method of generating heading lines is that you may get more heading lines than you want. Column names with underscores improve their readability.

### Floating Point Data

If you are reporting floating point data (scientific data), you will probably want to increase the Column Width specification so that you can see more than 3 digits of information. You can also use the Column Width specification to truncate low order information from TIME, DATE, and TIMESTAMP fields.

## PACKAGING YOUR QUERIES

Now that you know what you can do with queries and reports, it's time to figure out how to make it easy for your application user to run a query and what it takes to package a predefined query and report with your application. To accomplish these tasks, you will need to learn about Procedures, OS/2 commands, and building queries for export.

| DEPT NUMBER | JOB | EMPLOYEE NAME | SALARY |
|---|---|---|---|
| 15 | Mgr | Hanes | $20,659.80 |
| | | JOB TOTAL = | $20,659.80 |
| | Sales | Rothman | $16,502.83 |
| | | JOB TOTAL = | $16,502.83 |
| | | DEPT 15 TOTAL = | $37,162.63 |
| 20 | Mgr | Sanders | $18,357.50 |
| | | JOB TOTAL = | $18,357.50 |
| | Sales | James | $13,504.60 |
| | | Pernal | $18,171.25 |
| | | JOB TOTAL = | $31,675.85 |
| | | DEPT 20 TOTAL = | $50,033.35 |

**Figure 14-7** Report with Matched Break and Sort

## Procedures

Query Manager provides Procedures that allow you to combine commands and to eliminate the need to remember command syntax. They can be very useful in running predefined reports, particularly those you have customized by adding your own heading text or those containing variables. Other uses of Procedures are discussed in Chapter 15, "Customizing Query Manager."

The following is an example of a Procedure containing a single command to run a query using a customized report form:

```
/*  SALES_QUERY                              */
/*                                           */
/*    Written: Jan 8, 1989                   */
/*    Author: Elvis Presley                  */
/*                                           */
/*    This query is used to highlight        */
/*    record sales in the Memphis area.      */
/*                                           */
/*                                           */
'RUN QUERY SALES_QUERY (FORM=MYREPORT)'
```

```
DEPT                        EMPLOYEE
NUMBER    JOB                NAME                SALARY
------    ---                ---------           ------
    15    Mgr                Hanes           $20,659.80
                                             ----------
                          JOB TOTAL =         $20,659.80

          Sales              Rothman         $16,502.83
                                             ----------
                          JOB TOTAL =         $16,502.83
                                             ----------
                      DEPT 15 TOTAL =         $37,162.63

    20    Sales              James           $13,504.60
                                             ----------
                          JOB TOTAL =         $13,504.60

          Mgr                Sanders         $18,357.50
                                             ----------
                          JOB TOTAL =         $18,357.50

          Sales              Pernal          $18,171.25
                                             ----------
                          JOB TOTAL =         $18,171.25
                                             ----------
                      DEPT 20 TOTAL =         $50,033.35
```

The top lines are strictly for documentation purposes, only the first line of a procedure must contain a comment. You (or someone else) can then go to the Procedures Main menu and select this procedure to be run. This is an alternative to entering this command or finding the query then doing a **Run Using** and typing the report form name.

You can combine the query command with a print command in the procedure to have the report printed rather than viewing it. This would be done by placing the following commands in a procedure:

```
/*  SALES_REPORT                          */
/*                                        */
/*    Written: Apr 23, 1989               */
/*    Author: Roy Orbison                 */
/*                                        */
/*    This is a quarterly report on       */
/*    video rentals in Wink.              */
/*                                        */
'RUN QUERY SALES_QUERY (INTERACT=NO)'
'PRINT REPORT (FORM=MYREPORT)'
```

Assuming that you name the procedure SALESM, you can now put together an OS/2 command file to run the procedure. It might look like this:

```
CALL  QUERYMGR  /DAT:DATABASE  /AUT:VIDEOS  /RUN:SALESM
```

"Using OS/2 Commands" provides more details on how to build the command file. If your query or report form has variables, they can either be initialized by letting the Query Manager prompt the end user for a value or you can use a SET statement in the procedure to initialize the variables before the RUN QUERY statement.

## Using OS/2 Commands

You can construct a simple one-line OS/2 command file to invoke the Query Manager and have it run a predefined query and report. One of the things you probably noticed in the preceding example was the CALL QUERYMGR command. The QUERYMGR command file is provided so that you can invoke the Query Manager to run predefined procedures. The command file's parameters tell the Query Manager how to operate. These parameters can help reduce the selections required to get Query Manager started and running the function you want, thereby providing a quick path through the menu hierarchy ("tree") to run your predefined queries and reports. The command file parameters are discussed below.

1. **Database Name** The name of a database to open (use). This is the name of the database that contains the tables, queries, and reports that you want to use. The parameter is specified as follows:

   ```
   /DATABASE:databasename
             or
   /DAT:databasename
   ```

   The databasename parameter is 1–8 characters long. The database name can also be provided in the Query Manager profile. The command line parameter overrides the database name specified in the profile and is the preferred method of specifying the database.

2. **Qualifier** Helps determine the full name of the procedure to be run. It is also used to pick an ID for lists. The parameter is specified as follows:

   ```
   /AUTHID:qualifier
           or
   /AUT:qualifier
   ```

   The qualifier parameter is 1–8 characters long. Query Manager works with its own set of objects, which were discussed in "Query Manager Objects" in Chapter 13. A database and Query Manager object's name consists of two parts. The first (or leftmost part) is called the Qualifier (/AUT). The second part is what you see displayed in the Query Manager lists. The following terms are used synonymously with Qualifier: ID for Lists, AUTHID, Authorization ID, and Collection name.

When you create or reference objects, there is an implicit usage of the current ID for Lists. The current ID is shown in the Queries Main Menu title line. You can change the ID for Lists before Opening a new query or Importing a query. When the query is saved, that ID will be used to prefix the name of the query. This provides a way to use the same name for more than one query or other object and keep them contextually separate.

Query Manager carries out several steps when it starts:

- It creates an object qualifier (called ID for lists) and sets its initial value to the value in SQLUSER. If SQLUSER has no value, the value is set to NULLID.
- Next it checks to see if a profile has been specified as one of the startup command line parameters. If it has, the ID for lists in that profile (if not blank) becomes the ID for lists.
- Finally it checks to see if a qualifier has been specified as one of the startup command line parameters. If it has, the ID for lists is set to the command line qualifier parameter value.

3. **Procedure to Run** Indicates that a Query Manager Procedure is to be run. This procedure may contain any sequence of valid Query Manager commands. This can be used to avoid some of the startup steps and help a user who is unfamiliar with Query Manager. This parameter requires that either the Profile or Database parameters be used to determine which database is to be used. The parameter is specified as follows:

```
/RUNPROC:procname
          or
/RUN:procname
```

The procname parameter is 1–8 characters long and is the name of a Query Manager procedure that has been previously created.

4. **Rowbuffer Size** Indicates the size of the row buffer to use. This parameter determines the size of an in-memory buffer to hold data that is retrieved from a table. The buffer can store up to 64K bytes of data. Its purpose is to improve performance by keeping data in memory. It is most helpful when you are viewing reports and want to scroll backwards (that is, see the previous screens of a report). Increasing this value will not necessarily improve performance since it may cause extra system swapping to occur. Whenever the operating system needs real storage, something has to be swapped out. The system does not know how much of this buffer is in use and, therefore, must have room for all of it. You should measure your response time using the Query Manager before and after changing the Buffer size parameter. If you do not see reduced response times, lower the value. If specified, this parameter will override the Buffer specification provided in the Query Manager Profile.

The parameter is specified as follows:

```
/ROWBUFFER:nnnK
```

Note that nnnK is the number of K-bytes to be used for the buffer.

5. **Profile Name** Indicates the Query Manager profile to be used. The parameter is specified as follows:

```
/PROFILE:profilename
          or
/PRO:profilename
```

The profilename parameter is 1–8 characters long. A Query Manager profile contains default information that may make Query Manager easier to use for a beginning user. It also contains information to control the operation of functions like Import/Export of Lotus spread sheet data (see "Table Utilities" in Chapter 16).

## Building and Exporting Queries

This is the last topic to cover before putting it all together to build some predefined queries to package with your application.

You were advised to install your application into an existing database so that disk space could be minimized and data sharing could be maximized. To do this requires care in naming your database and Query Manager objects. One way of handling this problem is to utilize a unique Qualifier for your objects. You can do this in your application installation process by asking the end user for the name of an existing database and a qualifier to be used for storing your objects. The Qualifier can be thought of as your application's name. In the preceding example, VIDEOS is the application name that was used.

If you take this approach, you must do a little advanced work to handle the Query Manager objects. You must have created Query, Form (that is, report), and Procedure definitions on your own computer. Each of these must be exported manually with the Query Manager so that they can be subsequently imported by the end user. You need to document a list of file names and Query Manager object names for the end user to import these into their database one at a time. Since this is a manual process, it is best to have as few definitions as possible. It is important that the installer's instruction include steps to change the ID for Lists so that the Query Manager objects are stored under the desired qualifier.

If you want to have lots of predefined Query Manager reports, consider requiring a separate database just for your application. In this case, you can create your Query Manager definitions as well as your application tables in a database. You could create this database on your computer and then use the Query Manager Backup utility (see "Database Backup" in Chapter 15) to put a copy on diskette for distribution. Your application database can then be restored on the user's computer in a single step.

There is one more consideration. When you define a query on your computer and use Prompted Query, the names of the data tables are always qualified with whatever the Qualifier is when you build the query. This probably will not be the same Qualifier your end user will pick to use. This problem can be circumvented by converting your Prompted

Query to a SQL Query and then deleting the table name Qualifier altogether. In one of our examples the SQL that was generated looked like:

```
SELECT NAME, DEPT, SALARY, DEPTNAME
 FROM BILL.STAFF A, BILL.ORG B
  WHERE (DEPT =DEPTNUMB ) AND
   ( ( SALARY > 10000 ) )
    ORDER BY DEPT, NAME
```

This would be changed to:

```
SELECT NAME, DEPT, SALARY, DEPTNAME
 FROM STAFF A, ORG B
  WHERE (DEPT =DEPTNUMB ) AND
   ( ( SALARY > 10000 ) )
    ORDER BY DEPT, NAME
```

This adjustment is not required if you use a separate database for your application.

## CONCLUSION

If your primary needs are extracting information to answer ad hoc questions or if you have limited report output requirements, then Query Manager should be more than satisfactory. If you need more flexibility in formatting your report, you should consider writing your own program or purchasing a report writer package. Another alternative is to use Query Manager to extract the data using a query, saving the data in a temporary table, then exporting the data to an ASCII file for later processing by another program.

If you use predefined reports, carefully consider how you package the Query Manager definitions. The next chapter will consider the Query Manager functions to implement other aspects of the application. They too will have an impact on how you package your application.

# CHAPTER FIFTEEN

# Customizing Query Manager

## OVERVIEW

This chapter discusses the Panels, Menus, and Procedures functions provided by the Query Manager. These functions collectively are referred to as the *customization tools*. You can use them to construct a simple end-user application without programming in a language like C. The major purpose of this chapter is to help you determine when to use the customization tools.

The Panels function is the major tool provided to construct an application. You use the Panels function to define which database tables and columns are to be processed by your application. A panel also defines operations that allow the application user to view, edit, or search for table rows. The Query Manager displays your panel and handles the application user's input based on the panel's definition.

A panel provides a window into the data stored in a table. A panel definition allows you to do the following:

1. **Place table fields anywhere on the display screen**
2. **Edit information from two tables using one panel** One of the tables is the *controlling table* or *Root table*. The second table (called the Subtable) must be related to the Root table using a field that appears in both the Root and the Subtable. These fields are called the connecting columns. The value of the connecting column in the Root table's row is used to locate one or more rows in the Subtable.
3. **Place textual information anywhere on a panel** For example, you might want to include instructions on what data to enter in fields or label an input field.
4. **Display read-only fields from Lookup tables**
5. **Display fields computed from other fields**
6. **Edit or view selected fields from a Root or Subtable** Note that you need not edit all the fields in a table.

The Query Manager Procedures and Menus functions allow you to define how panels are combined to form your application. In addition, a procedure can be used to provide conditional logic based on the application user's input to display different panels or perform different actions.

The preceding paragraphs have attempted to provide you with a very brief overview of the Query Manager's customization tools. For a more complete description, you should read the *IBM OS/2 Extended Edition User's Guide* and follow the exercises provided in that book.

The next two sections of this chapter, "When to Use Panels" and "When Not to Use Panels," provide you with guidelines to help you determine when to use the Query Manager's customization tools. The last section of this chapter, "Using Panels," contains information supplementary to the *IBM OS/2 Extended Edition User's Guide* that you may find helpful when developing a Query Manager application.

## WHEN TO USE PANELS

Use the information on when and when not to use Panels simply as a set of guidelines for determining when to use the Query Manager Panels function versus writing your application in a language like C. You must take the guidelines and apply them to your situation and determine whether or not they apply. There is no simple, concise methodology for making this determination.

Most people with some programming experience can use the Query Manager customization tools to make an application. Some people without programming experience who follow the exercises provided in the *IBM OS/2 Extended Edition User's Guide* will be able to define an application with these functions. A simple application can usually be put together in a much shorter period of time than writing the application in a procedural language like C. However, an intimate knowledge of the tables, their data fields, and relationships is required especially when data editing is to be performed.

Although building applications without doing conventional programming sounds like a good idea, Query Manager Panels is not the answer to your dreams. It is, however, a tool that may be used to implement some applications. The first step, of course, is for you to determine if Panels is the tool to use to build your application.

You should consider using the Query Manager Panels function for application prototyping, a simple application, or a personal application. Each of these application types is discussed in more detail in the following sections. If you decide to use Panels, you should also read "Using Panels" at the end of this chapter.

### Application Prototyping

One technique for designing applications is to build a prototype. A prototype typically does not contain all the functions. It does, however, allow you to get a feel and touch for the application and its flow. To be useful, the prototype must be easy to modify and should be built with very little effort or time.

A Query Manager Panel application can be built more quickly than most applications written in a conventional programming language. The only drawback is that a Panels application may not be able to simulate the desired application function.

### Simple Applications

Simple applications have the following characteristics:

1. **Viewing Data** These applications do not provide a facility to update any information. A Panels application can only view one row at a time, but has the flexibility of pulling in information from Lookup tables associated with that row.

   A phone book application is an example of when you want to see multiple rows at once. The Prompted Query function comes closer to a solution for this type of application than does Panels.

2. **Minimal Updating** A simple update application does not have any data input restrictions (that is, editing requirements). It also updates no more than two tables. Single table updates can be done by simply editing the table using the editing functions provided in Tables and Views Menu. Panels gives you some flexibility in positioning the input fields on the screen. You can use this flexibility to make the screen match the data input form and thus make data entry and edit easier.

3. **Few "Pieces"** A simple application usually does not involve many pieces. In other words, you may have one or two Procedures, a Menu, and one or two Panels. As the number of application "pieces" increases, so does complexity and the need for function.

These are a few ways of describing a simple application. Few applications are simple given these restrictions.

### Personal Usage Applications

These are applications that you develop for your own private use; other people will not be using them. With these applications you need to understand only how the application is supposed to work, what data you will type into the fields, what action keys to use, and how to recover in error situations.

Unlike applications that others will use, applications for private use need not be very friendly and easy to use or have many error checking and recovery functions. Those types of functions are not easily provided via Panels.

## WHEN NOT TO USE PANELS

Since it is often easier to describe what something isn't rather than what it is, this section describes situations where using Panels may be more difficult than using a conventional programming language.

Check the topics in the following sections when evaluating the appropriateness of using Panels.

### Applications for Others

Applications that will be used by others have greater requirements for user friendliness, error checking, and recovery functions. In addition, you should consider the messages that another person will see. You cannot customize error messages the end user sees because they cannot go into an application context. Thus, the user will see Query Manager or SQL error messages rather than "The customer number you entered does not exist." End users who are unfamiliar with computers may not be able to use Panels.

### Data Concurrency

Since Panels provides a very limited capability for concurrency with other applications that use the same tables, this type of application is probably best designed and written in another language. Concurrency is limited because locks on the data are retained until control is returned to the Controlling Panel. Concurrency can become a problem if you plan on migrating your data and sharing it between multiple users in the future. The same Panels application run by multiple users against the same data will have this concurrency consideration. See "The Controlling Panel" in this chapter for more information.

### Exploiting the Computer

Panels does not let you exploit the functions of the computer hardware. For example, you might want your application to use the mouse and let the user select items from lists rather than keying data and using function keys. However, you will quickly find that the user must move from the pointing device to the keyboard, a disruptive practice. Therefore, the mouse is quickly discarded and with it a simpler way for the user to interact with your application.

Another disadvantage of Panels is that you can change neither the colors used nor the highlighting attributes for screen fields, both features that enhance information presentation. As a result, you may need to reduce the amount of information you put on a single display screen and reconsider how it is arranged. A Panels application can provide the end user only with a text-based interface; you cannot provide or build graphical information for display.

Finally, you may want the user to be able to resize the Panels window and look at some other information on the OS/2 Desktop. Unfortunately, Panels does not provide that capability, nor does it allow you to cut, copy, or paste information between it and a Presentation Manager application.

### Complex Applications

A Panel application that causes more than three levels of *nesting* is definitely complex. Nesting occurs when a Panel, Menu, or Procedure causes a subordinate Panel, Menu, or Procedure to be run. It is quite common for a Procedure or Menu to run (nest) a subordinate Panel. When this occurs, a new application environment is created and any locks acquired on tables by this and higher application levels remain while the nested object is run.

Three levels is an arbitrary number chosen based on the resources required and the implicit table locking that occurs. The Panel designer must be wary of these, particularly if the application is to share the tables with another application. Deadlock conditions can occur between applications if locking is not considered when sharing tables. This is typically the sort of thing a programmer is more trained to handle.

Another complexity issue is that very few tools exist to assist you in debugging a Panel application—all you can do is run it and look at the results. You can use the TRACE command to trace procedure statement execution, but it does not provide information relevant to Menus or Panels.

Query Manager provides limited error recovery. If every thing works correctly, your database changes are committed. If there is a deadlock or other SQL error indicated, a rollback is done. A rollback causes all table positions to be lost and the end user is forced to terminate the Query Manager application and restart it.

## Complex Data Editing

You can use Panels to do some field editing. Editing of the data entered by an end user is done by Database Services when the data is inserted or changed in the database. This type of data editing observes several limitations:

1. **Valid data format** You cannot enter non-numeric data in numeric fields. Also, TIME, TIMESTAMP, and DATE must be correct.
2. **Magnitude of numeric data** You cannot store a value of 100,000 in a small integer field.
3. **Field information required (non-null data)**
4. **Matching and range checking** The value selected or entered must match some entry in a defined table of values. For example, JOB_CODE must be in this list (22, 24, 27, 32, 35, EX). An example of range checking a numeric field is that its value is between 10 and 230. A character field example is for its value to be between "D" and "M." Both of these types of edits must be done by creating a view of the table and using the WITH CHECK OPTION and a WHERE clause. You can use the IN operator to specify a set of values that a field must match.
5. **A subrow field identified to Query Manager as a unique key** When a subtable that will be updated is specified, some field in that table is defined as being a unique key. Query Manager ensures that the values provided for that field are unique irrespective of whether the field is defined to Database Services as a index. Performance can be improved by defining these fields as unique indexes.
6. **A field identified to Database Services as a unique index**

This list may seem complete, but often there are other requirements. The following list is a set of editing requirements that are *not* handled by the Query Manager.

1. **Pattern edits** The first character must be alphabetic and the second through fourth positions must be numeric.
2. **Character folding** Fold all input to uppercase.
3. **Character substitution** Map the character input to some string. This provides a shorthand method for supplying the data in the format in which it exists in the database. It also makes list selection using mnemonics possible.
4. **Interfield edits** If JOB_CODE is 32, then the SALARY must be less than 50,000. DEBIT must be less than ACCOUNT_BALANCE.

Although there may be other editing capabilities not shown in this list, these items represent some common things that applications do. The only technique provided by Query Manager to do some of these is for you to define a function key that invokes a Procedure (you must write this) to do the editing. The end user has to press the edit function key and then press the update function key. If the end user chooses to ignore the editing, you will lose all application integrity of your data.

### Performance

As an interpretive form of programming, Panels has a few limitations. For example, Panels is not bound to machine instructions until a statement (or language directive) is executed, this is inherently slower than compiled languages. You may also find that Panels sometimes forces you to perform data manipulation or computations with several operations. Finally, to run a Panels application requires most of Query Manager to be in memory. A Query Manager Panel will usually require much more memory than writing the same application in a compiled language.

### Integrating Other Programs

There are no functions or exits that allow you to invoke another application to perform a function. Therefore, you cannot exploit a function that has already been written in another programming language or one that you might want to write to perform some function not provided by Query Manager.

This will increase your application development cost since it requires re-doing existing application functions.

### Database Design

You can access or update only two tables from a single Query Manager Panel. This may lead you to design tables that do not have a high degree of normalization. The five degrees of normalization are discussed in the *IBM OS/2 Extended Edition User's Guide*. Generally, as the degree of normalization increases, fewer fields are stored in a single table. Instead, tables are joined together to provide a composite of the information. Consequently, you may be forced into a less flexible database design if you use Query Manager Panels.

### Editing LONG VARCHAR Fields

You cannot use a Panel to edit a LONG VARCHAR field. If your tables contain any of these fields, you must create and maintain them with some other kind of application.

## USING PANELS

By now, you have reviewed your application requirements and perhaps have decided to use the Query Manager Panels function. If this is the case, you should also have read the *IBM OS/2 Extended Edition User's Guide* and done the exercises on customization (using Menus, Panels, and Procedures). Remember that this chapter is not a tutorial on Panels. Rather, it is supplemental information not provided in the *IBM OS/2 Extended Edition User's Guide* that may help you work with Panels.

### Structure

Applications can be described in terms of the business problem(s) you are trying to address, how they interact with the end user, or their structure (that is, the parts and how they relate to each other). This section discusses how a Query Manager application is structured. You build a Query Manager application (also called customizing) by arranging Procedures, Panels, and Menus into a hierarchical structure. Figure 15-1 shows this structure.

This diagram does not depict a specific application but it shows a relationship between pieces used to build typical Query Manager applications. Throughout the remainder of this chapter we use the term *customization object* to indicate the instance of a Panel, Menu, or Procedure. When you cause a Procedure, Menu, or Panel to be run, an instance of the object is created to represent its execution. For example, if Procedure A causes Procedure B to run, which causes Procedure A to run, there are two instances of Procedure A and one instance of Procedure B.

You usually start building an application with a Procedure. This Top Procedure does some initialization of global variables and passes control to a First Menu. Some applications are built starting with the First Menu. A Menu is usually used to ask the end user to select which major application function is to be performed. Based on the end user's choice, the Menu will then choose either to run a Panel or pick another Menu or a Procedure. A Panel is the major function provided to present data from tables and to provide input fields that can be used to change or add data.

If you have not read the *IBM OS/2 Extended Edition User's Guide*, you must define the contents of a Procedure, Menu, and Panel. Before defining the various Query Manager objects (we really could call them programs), you should decide what you want your application to do. You also need to determine the data (tables) with which your application will work. As you can see, designing the application is just as complicated with Query Manager as it is with any other conventional programming language. What the Query Manager customization functions can do, however, is reduce the implementation (coding) phase of building the application.

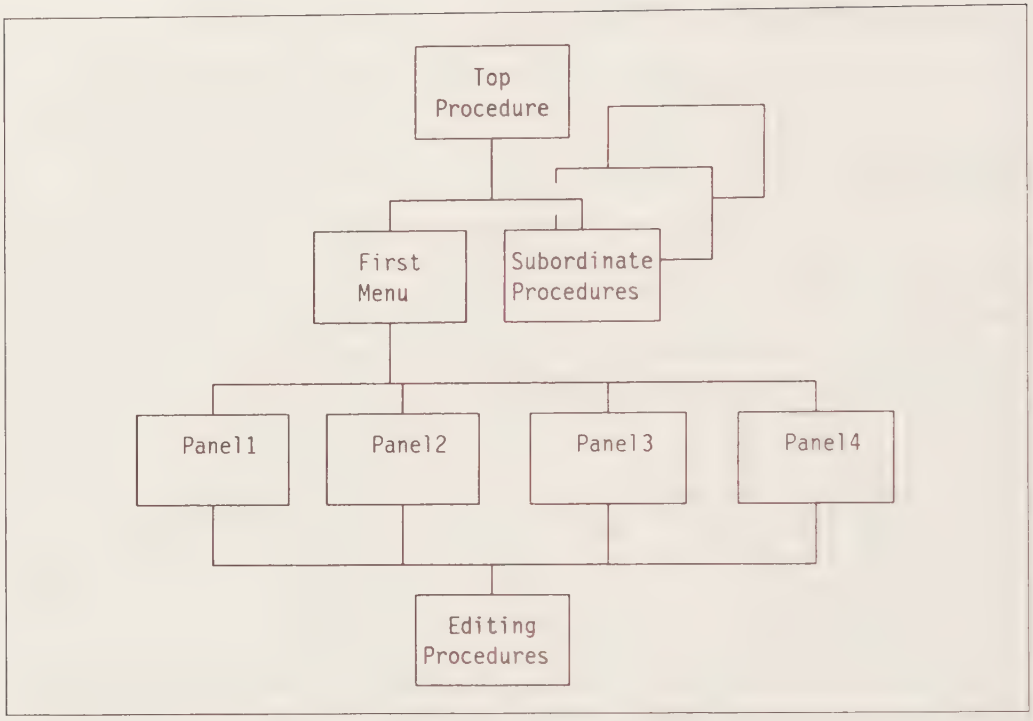



**Figure 15-1** Typical Query Manager Application Structure

To begin, define the application requirements and briefly describe how the application will interact with the end user. The end user usually directs the program to take some actions and the program asks the end user for data (input) needed to perform those actions. After you have determined the data tables to be updated, the sequence of tables to be updated, and the data editing requirements, a diagram like the one shown in Figure 15-1 will help you understand the Query Manager Panels, Menus, and Procedures that you must define to build the application.

## Variables

Procedures, Menus, and Panels all manipulate variables in a pool, which is the location in the computer's memory where the variables are stored. The two types of pools are the Procedure Variable Pool and the Query Manager Variable Pool.

The Procedure Variable Pool holds procedure variables. Procedure variables are not shared between Procedures, nor are they accessible by a Menu or a Panel. Each time a Procedure is run, a new instance of its variables are created. When the Procedure terminates, its variables are thrown away. A variable referenced in a Procedure that has not been set by a previous statement in that Procedure will cause the Query Manager to prompt the end user to supply a value for that variable.

The Query Manager Variable Pool holds Query Manager variables. Here are some examples of Query Manager variables.

1. All root and lookup table fields referenced in a Panel. Since variables are not indexed (there is no subscript mechanism), fields from a subtable with repeated rows cannot be referenced.

2. Procedure statements can create Query Manager variables using the SET command and read them using the GET command.

An instance of a Query Manager variable will be created at the level of the currently running object unless specifically created as a GLOBAL variable. Once created, a Query Manager variable is shared with any subordinate customization object. This mechanism passes parameters and allows customization objects to communicate.

The term *level* in the previous paragraph might be best explained by referring to Figure 15-1. If you look at the diagram, the level starting with the topmost block (Top Procedure) is one and increases by one as you go down the hierarchical tree. Therefore, when a new customization object is created (run), a level is created in the Query Manager Variable Pool. The level contains instance variables associated with the object. The table fields associated with PANEL1, PANEL2, PANEL3, and PANEL4 in Figure 15-1 are all at the same level and thus cannot exist at the same time. An object can access instance variables of a lower-level object. In our example, PANEL1 could could access an instance variable created by Top Procedure.

There is a level zero (called GLOBAL) of the Query Manager Variable Pool. Its variables must be explicitly created by the customization objects, except for the System Variables (DATABASE, TIME, and so on). Once a GLOBAL variable is created it can be changed but never deleted.

Procedures (particularly those used to edit or manipulate data) must first retrieve the Query Manager variable, operate on it, and then put it back. The end user can negate the work of an edit Procedure by simply keying a change and requesting the **Update** action because a Panel **Update** action can only be initiated by end user action. Update means a **Change and next** or **Add and next** Panel action. Table data is read from Database Services into the Query Manager Variable Pool. If the operator causes an Update operation to be performed, only those Panel fields indicated as output types are written back to the database from the Query Manager Variable Pool. As mentioned earlier, checking of the data is very limited when it is written to the database.

## The Controlling Panel

A Panel is a definition of the fields from a table that the end user views and may change. It is also a definition of the records (or rows in a table) the end user will see. The Panel selects one table row at a time that meets your selection criteria. For rows that are selected, only those data fields you want to look at are presented. Once a Panel starts, it moves sequentially through an entire table searching for rows that meet your search criteria.

A Panel can also define a Subtable that is to be processed along with this Root table. The Subtable is related to the Root table via a common field (called the *connecting column*) in both tables.

Many applications require more than two tables to contain their data. Multiple tables are defined to express the relationships between the application data. An application update (transaction) may involve changes to several tables. To accomplish this requires multiple Panels, perhaps one for each table. You can cause another Panel to be run by defining a panel or menu action to RUN PANEL. This can also be done from a procedure. This action (RUN PANEL) creates another execution environment in a hierarchical fashion (see "Variables" in this chapter). This is equivalent to adding another level of Panels onto the bottom of Figure 15-1. A new execution level of a panel is called a ***subpanel***.

The Controlling Panel is the panel at the lowest level and can be thought of as the first or original parent panel. If a group of panels are related in an application sense, you may want to consider using the BEGIN WORK and END WORK commands to ensure that none of the panel actions will work unless they all work. BEGIN END WORK lets you describe a panel and its subordinate customization objects as a single unit of work. All database changes are backed out if a failure occurs at any point. The negative side effect to this approach is that using these functions increases the number and duration of locks on the data. If concurrency is a concern, you should do the application with a language like C.

## CONCLUSION

The Query Manager Panels function can best address application prototyping, simple applications, and personal applications. You may choose to use it for other types of applications as you see fit. The key to using the Query Manager customization tools or an application developed in a conventional programming language successfully is the design of the data tables and how the application uses the Database Services's functions.

The next chapter explores how the Query Manager can be used to run the utility functions provided by Database Services. These utilities may be used to supplement your application and thus eliminate the need for you to write programs to interface to these Database Services functions.

# CHAPTER SIXTEEN

# Database Operational Support Tools

## OVERVIEW

This chapter focuses on the Query Manager tools you can use to maintain your database objects. These tools deal with either the database (as a whole) or with a table within a database. Whether or not you choose to rely on Query Manager to provide these functions is a choice based on the type of application you have and the skill level of the end user.

The Query Manager provides many useful database support tools. This chapter describes only those tools that most application developers believe supplement their application. For example, table creation and reorganization are not covered since most applications automate these functions and do not document user steps to perform them. You can learn more about these additional Query Manager tools either by using Query Manager or by reading the *IBM OS/2 Extended Edition User's Guide.*

The Operational Support tools are spread throughout the Query Manager menus. To some this may seem to be an odd organization. If you recall in "Query Manager Objects" in Chapter 13, the Query Manager user interface conceptual model takes actions on objects. Thus, you first select an object to use, then pick an action to be performed. If the model were action-object, you would pick the action first and then the object on which the action was to be taken. In an action-object model, all of the Operational Support tools would be located together.

## DATABASE UTILITIES

The database utilities are tools that deal with the database as an entity. In other words, these are actions that can be taken against a database object. The Query Manager provides tools to create, erase, catalog, backup, and restore a database.

The backup and restore tools are the ones you should consider as supplementing your application. These tools provide a simple way for a fairly unskilled user to back up the data stored in the database. Having some sort of data backup and recovery capability is a must for guarding against a disk media failure or a failed application that committed changes to the database in error.

You can also use backup to copy or move an entire database from one disk to another one (whether on the same or a different PC). Backup has several advantages if an entire database is to be manipulated:

- It is faster than exporting all your data tables and Query Manager objects.
- It does the backup for your application data as well as any other application's data at the same time.
- It is easier to use since it is a single step process.
- It avoids recreating views and rebinding access plans.

  You cannot export a view definition or an access plan. Exporting a view causes the result table to be exported. If this is in turn imported, it will then become a separate physical table (not a view of a table).

  Since access plans cannot be exported, the bind files associated with the programs that use a database must be copied (with the OS/2 COPY command, for example) so that the BIND utility (SQLBIND) can be executed to recreate the access plans for the receiving database.
- Most users can perform the Query Manager Backup and Restore.

To take advantage of Query Manager's Backup and Restore simply requires that you document how to use them. The process is described below.

### Database Backup

The **Backup** option in the **Tools** pull-down in the Query Manager Databases Main Menu (Figure 13-2) is used to back up a database. Once you have backed up an entire database, you can then choose to back up only those items that have changed since the previous backup. These are called incremental backups.

***Note:*** Although incremental backups are faster in most cases than backing up the entire database, they may be slower than expected since backup determines which OS/2 files making up the database have changed since the previous backup. Changing one field (column) in one row of one table will change at least one OS/2 file. If the field is used in one or more indexes, an additional file will be changed.

You must do the following steps before going to the action bar:

1. Select an existing database.
2. Make sure there are no other applications using that database. The Backup function opens a database for exclusive usage to prevent other applications from using the database and making modifications while the database is being backed up. If another application is using the database, you will get an error message.

When you start the Backup utility, the following steps occur.

1. The Query Manager prompts you for the database's password, the disk drive to hold the output, and whether you want all or only subsequent changes backed up. You must put the output of the backup on a different OS/2 drive. This may be inconvenient when you are taking the backup in case of an application failure; however, a backup on the same drive is worthless for recovery from media failures.

2. A blank screen replaces the Query Manager menu. Database Services uses the OS/2 Backup function (BACKUP) to copy the database's OS/2 files. BACKUP creates a blank screen to show messages indicating various files are being copied onto the backup disk. If the backup copy is on diskette, the OS/2 Backup utility prompts in this screen for diskettes to be inserted or removed as necessary. If the backup copy is placed on a fixed disk, BACKUP creates a subdirectory off the root named BACKUP.

   If you switch to the Task Manager window, you see an entry in the list created by Database Services that contains the name of the database when it was created. It looks like this:

   ```
   Backing Up Database database-name
   ```

   If the database name you select is not the same as the original name, the name shown is different. The name shown is the name of the database that was originally created.

3. The Query Manager menu reappears when BACKUP finishes.

### Database Restore

You can use the **Restore** option in the **Tools** pull-down in the Query Manager Databases Main Menu (Figure 13-2) to restore all of a database from your backup.

Before going to the action bar, do the following steps.

1. Make sure there are no other applications using any database. This will slow down the restore process. (It is even more prudent not to run *any* other applications while restoring a database.)

2. Create a database. Query Manager will restore only to an existing database. Therefore, you must create a new database (see "Create New Database" in this chapter) before requesting the Restore or overwrite an existing database. If you want to Restore and Create a database in a single step from a backup, you can write a utility program to do this using the Database Services APIs.

When you start the Restore, the following steps occur.

1. Query Manager prompts you for the new database's password.

2. Query Manager asks if you want to continue since the Restore will overwrite the database. Query Manager does not know whether an existing database should be destroyed

so that the Restore can replace it. This prompting may be a little cautious, but perhaps it avoids a user selection error.

3. A blank screen replaces the Query Manager menu. Query Manager uses the Database Services Restore function which in turn uses the OS/2 Restore function (RESTORE) to replace the database's OS/2 files. This screen shows messages indicating files are being copied onto the disk. If the backup copy is on diskette, RESTORE prompts in this screen for diskettes to be inserted or removed as necessary.

   If you switch to the Task Manager window you see an entry in its list created by Database Services that contains the name of the database being restored. It looks like this:

   ```
   Restoring Database database-name
   ```

4. The Query Manager menu reappears when RESTORE finishes.

The Query Manager Restore utility is useful for copying an entire database to another disk drive or another system. The database can be renamed after it is restored. Restore can also be used to recover a database whose password you have forgotten. You can only recover the database to the point of its last backup.

### *Create New Database*

A database is created so that you can store data or restore a backup copy of a database. The **Open** option in the **Actions** pull-down is used to create a new database. Before going to the action bar, select the ***---NEW---*** object in the database names list shown in the Query Manager Databases Main Menu (Figure 13-2).

When you **Open** a ***---NEW---*** database, Query Manager presents a pop-up that asks you to provide the items listed below.

1. **A name for the database** This 8-character name is subsequently used to refer to the database. The name is used both for the database's *reference name* and for its *original name*. The original name must be unique with respect to the volume on which the database is allocated. The disk **Drive** value determines the volume where the database is created. The original name never changes as long as the database exists. The reference name must be unique with respect to all the databases known to exist on this PC. This name can be changed by using both the Uncatalog and Catalog functions. There can be more than one reference name for the same database. A program (or Query Manager) uses a reference name to access the database.

2. **Password** (optional) Every time a database is opened or manipulated (for example, when another tool like Backup is run), the user is prompted for the password. If a password is not specified when you create the database, the database does not have one. Password prompting always occurs regardless of whether or not the database has a password. If the database does not have a password, simply press Enter to continue when prompted for the password.

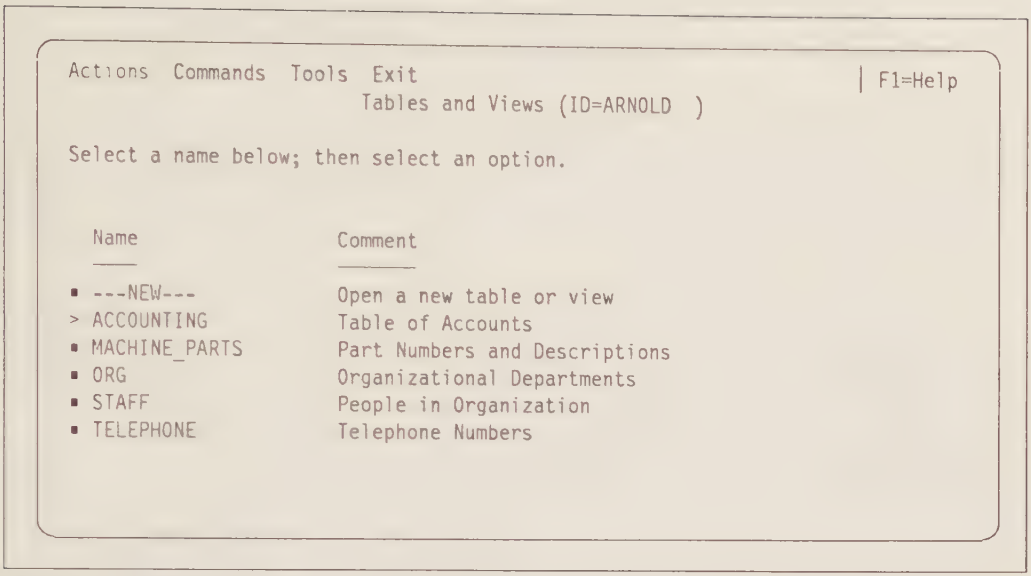



**Figure 16-1** Tables and Views Menu

3. **Disk drive** (optional) This input parameter determines where the database is to be located (allocated). If not specified, it is placed on the disk drive containing the Database Manager program libraries (the \SQLLIB subdirectory). This subdirectory is put on your C drive during installation. If you want to put a database on a diskette or a different fixed disk, then specify the OS/2 drive letter (A, B, D, and so forth) of the disk drive.

4. **Comment** (optional) This parameter can be used to store information you want displayed on the Query Manager Databases Main Menu just to the right of the database's name. It can be up to 30 characters in length. It is useful when the 8-character database name does not convey the purpose of a database. The comment can be replaced later by using the Uncatalog function followed by the Catalog function. Whenever you Catalog a database, you can provide a comment.

## TABLE UTILITIES

The Database Services utilities support creation and maintenance of the Database Services Objects (Tables, Views, and Indexes). If you need to copy data into or out of a database for use by another application, you can use the Query Manager Import and Export functions. For example, you may want to extract all or selected pieces of an inventory table and put the data into a format that a spread sheet application could use.

Import and Export are actions done on table objects. These tools can be accessed via the **Tools** actions on the Tables and Views Menu shown in Figure 16-1. The tools are easy

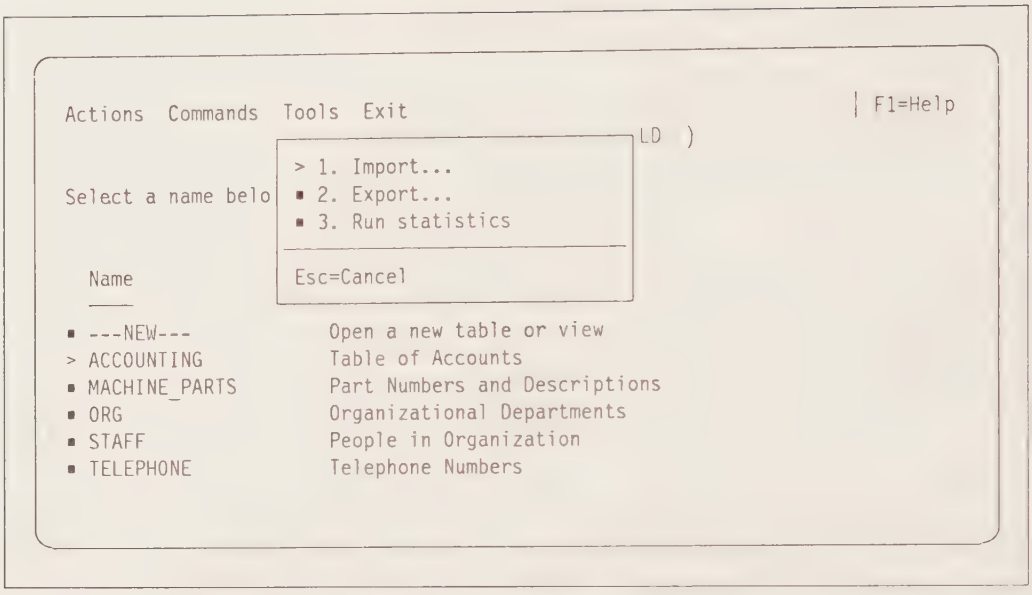



**Figure 16-2** Tables and Views Menu, Tools Pull-down

to use, and you may find that the Query Manager Import and Export tools can eliminate your writing additional application programs. How to work with these tools is covered below.

Figure 16-2 provides an idea of what the **Tools** pull-down menu looks like.

## Export

The Export utility provides several functions:

- Copies a table and its indexes to an OS/2 file
- Backs up a table
- Takes data from a table and puts it in a format acceptable to an application not written for SQL.

Export puts its output into an OS/2 file that you can name. The output file can be in one of the following formats:

**IXF** Integration eXchange Format (IXF) should be used if you want to back up or copy a table. There are several reasons for picking this format:

1. The table definition is included in the EXPORT file.
2. Index definitions are included in the EXPORT file.
3. Indexes are recreated during IMPORT of the table data.

4. Table and indexes can automatically be created during IMPORT.
5. Supports **Special data** (LONG VARCHAR).
6. Will use multiple diskettes as necessary to store the output file.
7. The export file also contains code page information about the character data. (Code page information defines how to interpret character information and allows it to be exchanged between users operating their PCs in a different language.)

The major advantage of IXF is that it is much easier to reconstruct the information being exported.

**DEL** Delimited format is provided for those users who have other applications (dBASE, for example) that can accept data in ASCII where each field is variable in length and is separated from the next field by a field separator character. This is sometimes referred to as Comma Separated Variable (CSV) format. Data in this format is usable by dBASE and programs written in BASIC.

Usually the field separator character is a comma (,). Character fields are delimited by a different character, usually a double quote ("). Records (rows from a table) are terminated by a carriage return plus line feed characters (CrLf). You can pick different characters to be used for the field separator and field delimiters in the Query Manager profile. You would need to do this when your character data contains double quotes.

Some applications cannot process more than 254 characters (the size of dBASE memo fields) per field in this format. Remember this limitation if you think you will need to copy the data for a non-database application,

**WSF** This is a term used for Work Sheet Format. It is specifically intended to provide data in a format that can be read by products supporting data in a format acceptable to Lotus Development Corporation's Lotus 1-2-3 or Lotus Symphony. More specifically, data can be exported in WKS, WK1, WRK, or WR1 formats. Where possible, character fields are translated from the IBM PC code page to the Lotus International Character Set (LICS).

WKS is the default format for WSF. If you want the data in WK1, WRK, or WR1 formats, you must create a Query Manager profile and make it active before you export the table.

It is *not* a good idea to specify a file extension for the output filename. If you do *not* specify the extension, Query Manager supplies the extension to match the format of the exported table. Therefore, you can check the generated file extension to ensure that it matches the format you need. For example, if you choose to export a table as WSF to a filename SPREAD1, Query Manager will create a file SPREAD1.WKS (assuming you are using the default Query Manager profile). If you change your profile to indicate WR1 format, the file created is named SPREAD1.WR1.

*Exporting*

The Query Manager provides two methods to do an Export:

1. **Prompted** Select an existing table object on the Tables and Views Menu, and then select the **Tools** actions. The pull-down menu provided (Tables and Views Menu, Tools pull-down) gives you an **Export** option. After selecting **Export**, you will be prompted to enter the output OS/2 filename and the format (IXF, DEL, or WSF).

2. **Command** The Query Manager EXPORT command syntax is

```
EXPORT TABLE tablename TO filename (DATAFORMAT=[IXF]
                                               [DEL]
                                               [WSF]

CONFIRM=[YES]  )
        [NO ]
```

You must specify the lowercase variables (that is, the names of the table and OS/2 files to be used). DATAFORMAT and CONFIRM are optional parameters. If they are specified, the filename must be followed with a left parenthesis "(" to indicate their presence. The valid operands for the optional parameters are shown inside the brackets. The first value in the brackets is assumed if the optional parameters are not specified.

The following is an example of exporting the ORG table to the OS/2 file C:\MYDIR\ORG.IXF in IXF format.

```
EXPORT TABLE org TO c:\mydir\org
```

The Query Manager export command can occur in any of the following contexts:

1. **The Query Manager command line** The Query Manager command line is a pull-down option from the **Commands** action. This action can be found on most of the Query Manager menus on the action bar.

2. **In a Procedure** Export commands and some other commands can be put together to form some useful process. Together these commands can be run as a Procedure. Using Export in a Procedure that is executed from a Panel should be avoided because of the potential for creating a deadlock condition with your Panel application.

3. **As a Panel or Menu Action** Export can be a Panel or Menu action. If the Panel or Menu is invoked from another Panel that is updating a table, Export should not be used because of the potential for creating a deadlock condition with your Panel application.

*Hints*

You may have applications that do not use the Database Manager with which you would like to share data from tables. EXPORT can be used to get a copy of data out of a table. Realize, however, that having multiple copies of the same data creates the problem of keeping all the copies updated. The information you might want to export for use with a non-database application might be one of the following:

1. **Only a subset of the information in a table** For example, you might want only two or three fields of data, or you may want only rows (records) where SALARY is greater than 50,000.

2. **Information from more than one table** When tables are designed, the data is usually split between tables. A relational database can then join tables together to provide the appearance of a single table containing all the data fields you are interested in.

3. **Computational variants** You may want to take the value of a field and perform some computation on it and use that as input to your other application. For example, you might want your budget application to assist you with some "what if" analysis. The application may do an excellent job of taking input and summing labor costs for all job categories, but it does not have the capability of adding 10 per cent salary increases and providing the result.

   In such a case you can use Query Manager to make a query that computes a field based on the value of a column in a table. For example, you can obtain the result of SALARY*1.1 (salary plus 10 per cent).

Let's look at this in a little more detail. You would take the following steps to export a selected subset of a table:

1. Determine what fields (columns) you want.

2. Use either the Query Manager Prompted Query or SQL Command line to create a view reflecting the data you want. Creating a view allows you to repeat the next step as often as you wish. How to create a view is explained after these steps.

3. Go to the Tables and Views Menu, select the view, and Export it to a file that can subsequently be read by your other application.

A view is a logical subset of the data in one or more tables and does not replicate the data that is in those tables. You can use views to limit the table data (columns and rows) that can be accessed or modified by a person or an application program. Views can also simplify the process of respecifying the data that is to be accessed. Suppose that you are going to make several iterative queries, all of which involve limiting the data retrieved to people with brown hair. It might make the job a little easier to create a view of only people with brown hair, then use the view to further constrain the information of interest for a particular query.

A view is easy to create with the Query Manager. Select the ***---NEW---*** object on the Tables and Views Menu and then select the **Open** option. Open asks you whether you want to create a Table or a View. If you create a view, you will be prompted to define a query that defines the view of a table. Query Manager creates a view and also stores the view definition so that it can later be displayed. Only views created by Query Manager Prompted View definition can be redisplayed later. (Redisplaying may be helpful if you want to see what the definition is or recreate the view on a different system.)

A view requiring one of the following SQL functions must be created by executing a SQL statement:

- Subquery
- UNION, INTERSECT, or EXCEPT
- Summary functions (AVG, SUM, etc.) in an expression
- Summary functions with DISTINCT
- HAVING clauses
- GROUP BY
- WITH CHECK OPTION. This option provides a limited degree of control over the values entered into a table field. For example, you could use this facility to ensure that Salary was always less than 60,000.

To specify these functions, use the **Command** action and select the **SQL Query** option. The SQL Query function allows you to enter and execute a SQL statement. The following is a SQL statement to create a view using the WITH CHECK OPTION:

```
CREATE VIEW myview (MY_NAME, MY_AGE, MY_SALARY, MY_DEPT) AS
 SELECT NAME, AGE, SALARY, DEPT FROM STAFF
        WHERE DEPT = 20
        WITH CHECK OPTION
```

It is a good idea to save this statement as a Query Manager query after it has been executed so that the statement can be reviewed or changed. The Query Manager cannot display a view definition created in this manner unless the query containing the create view is saved.

When you Export a table or view, an OS/2 file is created in the \SQLLIB subdirectory named QRWEXPRT.LOG. This file is used to hold the messages generated by the Database Services Export function. Query Manager will warn you if any errors occur during the export operation. If errors occur, you must switch to another OS/2 session and view this file to determine what may have happened. It is probably a good idea to use simply the PRINT command to print it out or use the OS/2 System Editor to look at it. The log contains a useful count of the number of rows processed.

If you use multiple copies of Query Manager, be aware that there is a single QRWEXPRT.LOG. Therefore, if you attempt concurrent exports one will fail since it cannot open the log. Another consideration is that the log is always overlaid (reused). Therefore, it is not a good idea to run Export from multiple Query Manager sessions since the last one will overlay the preceding export error messages.

### Import

The Import utility provides several functions:

- Creates a table and its indexes from an OS/2 file.
- Restores a table.
- Puts data in a table from a non-database application.

Import takes its input from an OS/2 file. You must specify its name. The input file can be in one of several formats. These formats were discussed in "Export," but some information is repeated in the following list.

**IXF** Integration eXchange Format (IXF) is the only format that can be used to create a ***---NEW---*** table. In other words, data in any of the other formats requires that you define (create) the table before doing the Import operation. There are several reasons for picking this format.

- The table definition step is done for you and avoids human error in the definition.
- Indexes are also defined and created during Import.
- Supports **Special data** (LONG VARCHAR).
- The Import file contains code page information about the character data.

The major advantage of IXF is that it is much easier to reconstruct the information being imported.

**DEL** Delimited format is provided for users who have other applications that create data in ASCII where each field is variable in length and is separated from the next field by a field separator character. This is sometimes referred to as Comma Separated Variable (CSV) format.

The field separator character is usually a comma (,). Character fields are delimited by a different character, usually a double quote ("). Records (rows from a table) are terminated by a carriage return plus line feed characters (CrLf). You can pick different characters to be used for the field separator and field delimiters in the Query Manager profile. You would need to do this when your character data contains double quotes.

**WSF** This is a term used for Work Sheet Format. It is specifically intended to define a data format for products generating data in a format compatible with the Lotus Development Corporation's Lotus 1-2-3 or Lotus Symphony products. More specifically, data can be exported in WKS, WK1, WRK, or WR1 formats. Where possible, character fields are translated from the Lotus International Character Set (LICS) to the Database Manager code page.

WKS is the default format for WSF. If the data is in WK1, WRK, or WR1 formats, you must create a Query Manager profile and make it active before you import the table.

It is *not* a good idea to specify a file extension for the input filename. If you do *not* specify the extension, Query Manager will supply the extension to match the format it expects to read as input. Therefore, Query Manager will fail to find the input file rather than try to import data in an unexpected format. For example, if you import a table as WSF from the file SPREAD1, Query Manager assumes you are using the default Query Manager profile and attempts to read a file SPREAD1.WKS. If you change your profile to indicate WR1 format, the input file read is named SPREAD1.WR1.

**ASC** This is another format provided to accommodate non-database applications that create files in a fixed ASCII format. If you choose this format, you will be prompted to supply a mapping from the input file record to each table column. Mapping means you define the byte position in the input file of where each field starts and ends. You need not map all the table's fields. The mapping definition itself can be stored as a database table so that it can be reused. This particular format option can import but not export data.

### *Importing*

Unless you have IXF format data, you must create the table before you can import. The Import utility, like the Export utility, can be run in either a prompted or command mode.

To use prompts, select an existing table object on the Tables and Views Menu, then select the **Tools** actions. The pull-down menu provided (Tables and Views Menu, Tools pull-down) gives you an **Import** option. After selecting **Import**, you will be prompted to enter the input OS/2 filename and the format (IXF, DEL, WSF, or ASC).

A Query Manager IMPORT command can be issued in a Procedure, from the Query Manager command line, or as an action in a Panel or Menu. The command syntax is

```
IMPORT TABLE tablename FROM filename (DATAFORMAT=[ IXF          ]
                                                 [ DEL          ]
                                                 [ WSF          ]
                                                 [ ASC          ]
                                                 [ FILESPEC=    ]
                                                 [  tabname2    ]
```

You must specify the lower case variables (the names of the table and OS/2 files to be used). DATAFORMAT is an optional parameter. If DATAFORMAT is specified, the filename must be followed with a left parenthesis "(" to indicate its presence. The valid operands for DATAFORMAT are shown inside the brackets. If not specified, IXF is assumed.

The following is an example of importing the ORG table from the OS/2 file C:\MYDIR\ORG.IXF in IXF format:

```
IMPORT TABLE org FROM c:\mydir\org
```

If DATAFORMAT = ASC is chosen, then the FILESPEC = parameter is also required.

FILESPEC identifies the name of a table that contains the input file mapping definition and must have previously been defined.

The Query Manager IMPORT command can occur in several contexts. Considerations for its usage are identical to the EXPORT command.

### *Hints*

Query Manager Import will always replace (or create) the contents of a table. You may need to append rows to a table instead of replacing them. Your options are to write a program or to be creative. The Database Services Import API provides the function to append rows to a table during import. You could write your own program that uses this API.

Taking the creative option means that you will use the Query Manager and the SQL Query option of the **Commands** action and follow these steps:

1. Create a duplicate definition of the table you want to append to. Call it TABLE2.
2. Import the data into TABLE2.
3. Go to the SQL Query option of **Commands.**
4. Type and run the statement

```
SELECT * FROM TABLE1
  UNION ALL
    SELECT * FROM TABLE2
```

5. When the Report is displayed, Go to the Command Line option of **Commands.**
6. Type and run the statement

```
SAVE DATA AS TABLE3
```

You cannot use SAVE DATA AS on a table with a **Special data** (LONG VARCHAR) column. You also lose the column names by using the UNION operator. The names can be saved by exporting and importing the data back into one of the original tables. Being creative isn't always the easiest thing to do.

Another creative procedure uses the Query Manager and the SQL Query option of the **Commands** action. Follow these steps:

1. Create a duplicate definition of the table you want to append to. Call it TABLE2.
2. Import the data into TABLE2.
3. Go to the SQL Query option of **Commands.**

4. Type and run the statement

```
INSERT INTO TABLE1 (col-name1,
                    col-name2,
                    col-name3,
                      .
                      .
                    col-namen)
    SELECT * FROM TABLE2
```

This solution cannot copy columns with **Special data** (LONG VARCHAR). It does retain column names, although you must type them into the column names list of the INSERT statement.

Because all the insert activity is written to the log file, you cannot use this solution when appending substantial data to a table. When the log file is full, the statement fails and all of the inserts are backed out of the table. In some cases, increasing the size of the log file may provide enough space for your statement to complete successfully. The Import utility itself will recover from a log full condition by COMMITing the inserts when the log becomes full and then continuing.

When you import a table, an OS/2 file is created in the \SQLLIB subdirectory named QRWIMPRT.LOG. This file holds the messages generated by the Database Services import function. Query Manager will warn you if any errors occur during the import operation. If errors occur, you must switch to another OS/2 session and view this file to determine what happened. It is probably a good idea to simply use the PRINT command to print it out or use the OS/2 System Editor to look at it. The log contains a useful count of the number of rows processed.

If you use multiple copies of Query Manager, be aware that there is a single QRWIMPRT.LOG. Therefore, if you attempt concurrent imports, one will fail since it cannot open the log. Another consideration is that the log is always overlaid (reused). Therefore, it is not a good idea to run import from multiple Query Manager sessions since the last one to run will overlay the preceding export error messages.

If you are importing substantial data, you may get a warning message that errors occurred during import. When you look at the log you will see the following message sequence:

```
SQL3186W Data was not imported into the database because the
         log was full. Recovery will be attempted. SQLCODE
         "-964" was returned.

SQL0964C The transaction log for the database is full.
```

Don't be frightened. Import inserts rows into a table. The database recovery log is not infinite and after some point it signals that it is full. At this point, Import will COMMIT its inserts and thus free up the log space and continue its work. If it fails to recover, you will get additional error messages and it will stop. You will aggravate the situation by running concurrent database applications while doing an Import. Log space is not freed

until all tasks reach a quiesced state. See "How Many Databases?" in Chapter 6 for more information.

Another common error situation is found when Importing delimited data where the character data contains the character delimiter (double quotes ("")). If this happens, error message SQL3114W occurs, and the data going into the field for a particular row is truncated at the point the imbedded delimiter is found. Input records containing erroneous field separator characters may go undetected and cause fields to be put in the wrong table columns.

### Importing Data from DB2 or SQL/DS

You can use the SQLQMF command to get a table from DB2 or SQL/DS that has been exported using the IBM Query Management Facility (QMF). SQLQMF uses the OS/2 Extended Edition Communications Manager to copy the data from the S/370 computer to the PC and then convert it into the DEL format used by the Import utility.

After using the SQLQMF command, the following steps can be done with the Query Manager:

1. Create a table to hold the table. If the table already exists, this step is unnecessary.
2. Use the Import utility to bring the data into the table.

The SQLQMF command may not provide the functions you need if any of these conditions exist:

- Your host files have a different naming convention.
- Your VM host data is not on your "A" minidisk.
- You do not have enough space on your C drive to hold the file.

Since this function is implemented with OS/2 command files, you can copy the CMD files and edit them to accomplish the previously mentioned functional limitations. A familiarity with QMF, the OS/2 Communications Manager, your host system, and OS/2 command files is required.

## CONCLUSION

This chapter has described those Query Manager tools you should consider using to maintain your database objects and concludes our discussion of the Query Manager. Query Manager is more than its name implies. Its primary purpose is to provide a user interface to support the Database Services utilities and configuration functions. It is also a query and report writing tool that can be used to extract information from your databases. Finally, Query Manager provides functions to construct simple applications without writing application programs.

We hope this book will enable you to more easily implement applications using Database Manager. Good luck and may all your tables be normal-ized.

# *Appendix A*

# Database Services C Function Names

Each of the Database Services function calls is listed in this appendix. They are listed according to the category to which each belongs.

## ENVIRONMENT ROUTINES

Database Connection:

**sqlestrd** Start using a database

**sqlestpd** Stop using a database

**sqlerest** Restart a database connection

Application Environment:

**sqleisig** Install default Ctrl-Break signal handler

Database Directory Manipulation:

**sqlecatd** Add database to system directory

**sqleuncd** Remove database from system directory

**sqledops** Open scan of database directory (system or volume)

**sqledgne** Get next entry from directory

**sqledcls** Close scan of database directory

Starting/Stopping Database Services:

**sqlestar** Start Database Services

**sqlestop** Stop Database Services

Database Creation:

**sqlecred** Create database

**sqlealtd** Change database password

**sqledchg** Change comment associated with database

**sqledrpd** Erase (drop) database

## UTILITY ROUTINES

Back up/Restore Database:

**sqluback** Back up database

**sqludres** Restore database

Data Exchange:

**sqluexp** Export data

**sqluimp** Import data

Performance:

**sqlureor** Reorganize table

**sqlustat** Generate statistics

## CONFIGURATION ROUTINES

Database Manager System Configuration:

**sqlfxsys** Get system configuration parameter

**sqlfusys** Update system configuration parameter

**sqlfrsys** Reset system configuration parameter

Database Configuration:

**sqlfxdbc** Get database configuration parameter

**sqlfeudb** Update database configuration parameter

**sqlfrdbc** Reset database configuration parameter

## APPLICATION SUPPORT ROUTINES

**sqlabind** Generate application access plan

**sqlaintp** Retrieve error message text

**sqlaintr** Interrupt the processing of a database request

# *Appendix B*

# Sample Programs

The sample programs are listed here in alphabetical order for your reference. Besides serving as examples of use of SQL and the Database Services function calls, these programs comprise a toolkit of command line routines for performing common database development tasks.

There are sample programs for creating and deleting test databases (CREATEDB.SQC and ERASEDB.SQC), importing data (DBIMPORT.SQC), and examining the database directory (DBLIST.SQC, RENAMEDB.SQC, CATALGDB.SQC and UNCATDB.SQC).

There are also programs that demonstrate the use of SQL statements. The most important of these are QUERY.SQC, which is a detailed example of the use of dynamic SQL, and the two programs that are an example of an application installation process (INSTALL.SQC and FILLTB.SQC). A fourth SQL program, DBSTRUCT.SQC, demonstrates the use of static SQL to build a general database utility that lists the tables and columns defined in a database. This program also demonstrates how a program can automate the binding process.

To work with these programs, simply create them on your system with an ASCII editor such as the OS/2 System Editor (E.EXE) and follow the steps in the module prolog for precompilation, compilation, and link. Details on this process can be found in Chapter 11, "Precompiling, Compiling, and Linking," including an OS/2 command file that performs the precompilation, compilation and link steps.

# CATALGDB.SQC

## Create a New System Directory

```
/************************************************************************/
/*  PROGRAM NAME: catalgdb.sqc                                          */
/*                                                                      */
/*  PURPOSE: Create a new system directory entry for a database         */
/*                                                                      */
/*  FORMAT: catalgdb drive dbname                                       */
/*                                                                      */
/*  INPUT:   drive = A-Z     -> file system on which the db exists      */
/*                                                                      */
/*           dbname          -> name used when the database was         */
/*                              created (8 char max)                    */
/*                                                                      */
/*  PROMPTS FOR:                                                        */
/*                                                                      */
/*           [alias]         -> optional database alias (8 char max)    */
/*                                                                      */
/*           [comment]       -> optional comment (30 char max)          */
/*                                                                      */
/*  OUTPUT:  The system directory is updated                            */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary                                              */
/*                                                                      */
/*  NOTES:   Database Manager does not have to be started for this      */
/*           program to run.                                            */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                 constant definitions                                 */
/************************************************************************/

#include <stdio.h>                      /* Standard C I/O functions     */
#include <string.h>                     /* C string manipulation fns    */

#include <sqlenv.h>                     /* SQL Environment include      */
#include <sql.h>                        /* SQL constants & err msg fn   */

/************************************************************************/
/*                 Global data elements                                 */
/************************************************************************/

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area      */
unsigned char alias[9] = "";
unsigned char type_string[1] = "";
unsigned char nodename[9] = "";
unsigned char comment[31] = "";
```

# CATALGDB.SQC (CONT'D)

```
/*****************************************************************/
/*                 forward declare of routines                   */
/*****************************************************************/

void errmsg(void);                      /* error message routine      */
void helpmsg(void);                     /* print out user help        */

 /*****************************************************************/
/*                    M A I N   R O U T I N E                    */
 /*****************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{

int rc;
unsigned char type = '0';

  if ( (argc >= 2 && *argv[1] == '?') || (argc < 3) )
     {
     helpmsg();                         /*   print out help           */
     return(0);                         /*   leave program            */
     }

  printf("Enter database reference name; (press 'Enter' for default): ");
  gets(alias);

  printf("Enter database comment: ");
  gets(comment);

  rc = sqlecatd(argv[2],alias,type,nodename,*argv[1],comment,0,&sqlca);

  if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)       /* error???        */

      errmsg();                                    /* yes - report it */

  else
      printf("\nDone!\n");
 return(rc);
}  /* end main */

/*****************************************************************/
/*                          E R R M S G                          */
/*****************************************************************/
void errmsg()
{

#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];

  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);  /* get error msg text      */
  printf("\n%s\n",buffer);                /* print msg text          */
  return;

}   /* end errmsg */
```

# CATALGDB.SQC (CONT'D)

```
/************************************************************************/
/*                          H E L P M S G                               */
/************************************************************************/
void helpmsg()
{

  printf("CATALGDB creates a system directory entry for a database.\n\n");
  printf("FORMAT:  CATALGDB drive database\n\n");
  printf("WHERE:   drive  = A-Z  -> file system that contains
         the database\n");
  printf("         database      -> the name used when the database
         was created\n\n");
  printf("PROMPTS FOR: reference name -> defaults to database\n");
  printf("             comment        -> optional comment\n\n");

  return;

}  /* end helpmsg */
```

# CREATEDB.SQC

## Create a New Database

```
/************************************************************************/
/*  PROGRAM NAME: createdb.sqc                                          */
/*                                                                      */
/*  PURPOSE: Create a new database                                      */
/*                                                                      */
/*  FORMAT: createdb drive dbname                                       */
/*                                                                      */
/*  INPUT:   drive          -> file system on which to create db        */
/*                                                                      */
/*           dbname         -> database name (up to eight characters)   */
/*                                                                      */
/*  PROMPTS FOR:                                                        */
/*                                                                      */
/*           [password]     -> optional password (8 char max)           */
/*                                                                      */
/*           [comment]      -> optional comment (30 char max)           */
/*                                                                      */
/*  OUTPUT:  database created on specified file system                  */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary                                              */
/*                                                                      */
/*  NOTES:   Database Manager must be started before execution          */
/*           of this program.                                           */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                  constant definitions                                */
/************************************************************************/

#include <stdio.h>                      /* Standard C I/O functions    */
#include <string.h>                     /* C string manipulation fns   */

#include <sqlenv.h>                     /* SQL Environment include     */
#include <sql.h>                        /* SQL constants & err msg fn  */

/************************************************************************/
/*                  Global data elements                                */
/************************************************************************/

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area     */
unsigned char password[9] = "\0\0\0\0\0\0\0\0";
unsigned char comment[31] = "";

/************************************************************************/
/*                  forward declare of routines                         */
/************************************************************************/

void errmsg(void);                      /* error message routine       */
void helpmsg(void);                     /* print out user help         */
```

# CREATEDB.SQC (CONT'D)

```
/*****************************************************************/
/*                    M A I N   R O U T I N E                    */
/*****************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{

int rc;

  if ( (argc >= 2 && *argv[1] == '?') || (argc < 3) )
     {
     helpmsg();                          /*   print out help          */
     return(0);                          /*   leave program           */
     }

/*   create the requested database                                    */

  printf("Enter database password; (press 'Enter' for none): ");
  gets(password);

  printf("Enter database comment: ");
  gets(comment);

  rc = sqlecred(argv[2],*argv[1],password,comment,0,&sqlca);

  if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)       /* error???        */

      errmsg();                                    /* yes - report it */

  else

      printf("\nDone!\n");

 return(rc);

}  /* end main */

/*****************************************************************/
/*                          E R R M S G                          */
/*****************************************************************/
void errmsg()
{

#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];


  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);  /* get error msg text      */

  printf("\n%s\n",buffer);                /* print msg text          */

  return;

}   /* end errmsg */
```

# CREATEDB.SQC (CONT'D)

```
/*****************************************************************/
/*                          H E L P M S G                        */
/*****************************************************************/
void helpmsg()
{

  printf("CREATEDB creates a database on the requested file system.\n\n");
  printf("FORMAT: CREATEDB drive database\n\n");
  printf("WHERE:   drive  = A-Z  -> file system to contain the database\n");
  printf("         drive  = \"\"   -> create database on file system where\n");
  printf("                           the Database Manager was installed\n");
  printf("         database      -> name for the new database\n\n");
  printf("PROMPTS FOR: optional comment and password\n");

  return;

}  /* end helpmsg */
```

# DBIMPORT.SQC

## Import File into Requested Database

```
/************************************************************************/
/*  PROGRAM NAME: dbimport.sqc                                          */
/*                                                                      */
/*  PURPOSE: Import file into requested database. Optionally create     */
/*           an index on a requested column.                            */
/*                                                                      */
/*  FORMAT: dbimport dbname impfile type option table                   */
/*                                                                      */
/*  INPUT:   dbname          -> database name (up to eight characters)  */
/*           import file     -> name of file containing import data     */
/*           type            -> import data format type                 */
/*           replace opt     -> replacement option                      */
/*           target table    -> table name                              */
/*                                                                      */
/*  PROMPTS FOR:                                                        */
/*                                                                      */
/*           [password]       -> optional password (8 char max)         */
/*           [index creation]-> optional index to be created            */
/*                                                                      */
/*  OUTPUT:  data file imported to specified table. Statistics          */
/*           recalculated for the target table & its indexes            */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               *
/*                                                                      */
/*  BIND:    Not necessary, automatic when executed                     *
/*                                                                      *
/*  NOTES:   Database Manager must be started before execution          */
/*           of this program.                                           */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                  constant definitions                                */
/************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#include <time.h>
#include <sql.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sqlcodes.h>

EXEC SQL INCLUDE sqlca;
```

# DBIMPORT.SQC (CONT'D)

```
 ****************************************************************************/
/*                  Global data elements                                    */
/****************************************************************************/

#define BUFSIZE 512
#define WIDTH 75
#define NOTFOUND SQL_RC_W100

unsigned char *step;                /* pointer to program step text   */

unsigned char msgbuf[BUFSIZE];      /* place for returned error msg   */

unsigned char password[9]="";       /* database password              */

int errorlvl = 0;

/****************************************************************************/
/*             Error Handling Options For SQL Statements                    */
/****************************************************************************/

EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL WHENEVER SQLWARNING CONTINUE;

EXEC SQL WHENEVER NOT FOUND CONTINUE;

 ****************************************************************************/
/*                    SQL Declaration Section                               */
/****************************************************************************/

EXEC SQL BEGIN DECLARE SECTION;

unsigned char prep_string[2000]="";     /* for CREATE INDEX STATEMENT */

EXEC SQL END DECLARE SECTION;

 ****************************************************************************/
/*                  forward declare of routines                             */
/****************************************************************************/

void errmsg(void);                      /* error message routine      */
void helpmsg(void);                     /* print out user help        */

/****************************************************************************/
/*                           MAIN ROUTINE                                   */
/****************************************************************************/

int main(argc, argv)
int argc;
char *argv[];
{

unsigned char database[9];          /* database name                  */
unsigned char imp_file_name[256];   /* name of OS/2 import data file  */
unsigned char imp_file_fmt[4];      /* import file format             */
unsigned char imp_how[50];          /* replacement options            */
unsigned char table_name[28];       /* target table name              */

struct sqldcol *mydcoldata;
struct sqlchar *mytcolstrg;
struct sqlchar *myfiletmod;
```

# DBIMPORT.SQC (CONT'D)

```
int rc = 0;

  if (argc >= 2 && *argv[1] == '?')
     {
     helpmsg();                          /*   print out help           */
     return(0);                          /*   leave program            */
     }

  sqleisig(&sqlca);              /* Install Signal Handler              */

/*****************************************************************/
/*  Start Using the Database                                     */
/*****************************************************************

step = "Start Using Database";

if(argc >= 2)                           /* argument 1 exists         */
  strcpy(database,argv[1]);
else
  {
  printf("Enter Database Name: ");
  gets(database);
  }
sqlestrd(database,password,SQL_USE_SHR,&sqlca);

/* if a password exists (assumed null password failed) ...           */

if (sqlca.sqlcode == SQLE_RC_INVPSW)
   {
   printf("Enter database password: ");          /* ask for password*/
   gets(password);

   sqlestrd(database,password,SQL_USE_SHR,&sqlca);/* try again       */
   }

/* if a restart required ...                                         */

if (sqlca.sqlcode == SQLE_RC_DB_RESTART)
   {
    printf("Database Restart in progress. Please wait . . .\n");

    step = "Restart Database";
    sqlerest(database,password,&sqlca);
   }

if (sqlca.sqlcode != SQL_RC_OK)         /* if sql error              *
   goto exit_error;                     /* go to error handling      */

/*****************************************************************/
/*  Process IMPORT Command Line Options                          */
/*****************************************************************/

/* set up import file name                                           */

if(argc >= 3)                           /* argument 2 exists         */
  strcpy(imp_file_name,argv[2]);
else
  {
  printf("Enter Import File Name: ");
  gets(imp_file_name);
  }
```

# DBIMPORT.SQC (CONT'D)

```
/* set up import file type                                              */

if(argc >= 4)                          /* argument 3 exists            */
  strcpy(imp_file_fmt,argv[3]);
else
  {
  printf("Enter Import Data Type (IXF, WSF, DEL): ");
  gets(imp_file_fmt);
  }

/* set up import option type                                            */

if(argc >= 5)                          /* argument 4 exists            */
  strcpy(imp_how,argv[4]);
else
  {
   if (0 == strcmpi(imp_file_fmt,"IXF"))
     printf("Enter Option(INSERT,REPLACE,CREATE,REPLACE_CREATE): ");
   else
     printf("Enter Option(INSERT or REPLACE): ");
   gets(imp_how);
  }

/* set up target table name                                             */

if(argc >= 6)                          /* argument 5 exists            */
  strcpy(table_name,argv[5]);
else
  {
  printf("Enter Fully Qualified Table Name: ");
  gets(table_name);
  }

/************************************************************************/
/*  Prepare the Structures for the Call to IMPORT                       */
/************************************************************************/

 /* choose columns from OS/2 file via Defualt method                    */

 mydcoldata = (struct sqldcol *)malloc(5);
 mydcoldata->dcolmeth = 'D';

 /* Build option string in mytcolstrg - option INTO table               */

 mytcolstrg = (struct sqlchar *)malloc(50);

 strcpy(mytcolstrg->data,strcat(strcat(imp_how, " INTO "),table_name));
 mytcolstrg->length = strlen(mytcolstrg->data);

 /* Build additional information string in myfiletmod                   */

 myfiletmod = (struct sqlchar *)malloc(30);

 if (0 == strcmpi(imp_file_fmt,"DEL"))
    strcpy(myfiletmod->data, "coldel, chardel\" decpt.");

 if (0 == strcmpi(imp_file_fmt,"IXF"))
    strcpy(myfiletmod->data, "FORCEIN");

 if (0 == strcmpi(imp_file_fmt,"WSF"))
    strcpy(myfiletmod->data, "");
```

# DBIMPORT.SQC (CONT'D)

```
myfiletmod->length = strlen(myfiletmod->data);

/* Do the import                                                    */

step = "Import";
printf("\nBegin Import\n");
sqluimp(database,password,imp_file_name,mydcoldata,mytcolstrg,
        imp_file_fmt,myfiletmod,"DBIMPORT.MSG",0,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)
   goto exit_error;

/*********************************************************************
/*  Bind ourselves to the database                                   */
/*********************************************************************

sqlabind("dbimport.bnd",database,password,"",NULL,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)
   goto exit_error;

/*********************************************************************
/*  Allow User to Execute SQL statments - i.e. for index creation    */
/*********************************************************************/

printf("\n");
printf("Enter Indexes for Creation. Blank to Continue.\n");
printf("i.e. CREATE INDEX index_name ON table (col1 [ASC|DSC], col2 ...)\n\n");

printf("Statement: ");
gets(prep_string);

while (prep_string[0] != 0)
   {

   EXEC SQL EXECUTE IMMEDIATE :prep_string;

   if (sqlca.sqlcode != SQL_RC_OK)   /* error on execute           */
      errmsg();

   printf("Statement: ");
   gets(prep_string);
   }

/*********************************************************************
/*  Run Statistics on the Table & All Its Indexes                    */
/*********************************************************************/

step = "Run Statistics";
printf("\nBegin Run Statistics\n");
sqlustat(table_name,0,NULL,SQL_STATS_BOTH,SQL_STATS_CHG,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)      /* error on run statistics     */
   errmsg();

/*********************************************************************/
/*  Normal program exit                                              */
/*********************************************************************/

printf("\nDone! Rebind any applications that use this database.\n");
```

# DBIMPORT.SQC (CONT'D)

```
exit:
  sqlestpd(&sqlca);

  return(errorlvl);

/*****************************************************************/
/*           E N D   O F   M A I N   P R O G R A M               */
/*****************************************************************/

/*****************************************************************/

/*****************************************************************/
/*                 ABNORMAL EXIT                                 */
/*****************************************************************/

exit_error:

  errmsg();

  goto exit;
}

/*****************************************************************/
/*                          E R R M S G                          */
/*****************************************************************/

void errmsg()
{

#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];


  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);  /* get error msg text      */

  printf("%s\n",buffer);                  /* print msg text          */

  errorlvl=2;

  return;

}   /* end errmsg */

/*****************************************************************/
/*                          H E L P M S G                        */
/*****************************************************************/

void helpmsg()
{

  printf("DBIMPORT: Import data.\n\n");
  printf("FORMAT: dbimport database impfile type option table.\n\n");
  printf("WHERE:  database    -> target database.\n");
  printf("        impfile     -> OS/2 file name for import data.\n");
  printf("        type        -> IXF, WSF, or DEL.\n");.
  printf("        option      -> INSERT, REPLACE, CREATE, REPLACE_CREATE.\n");
  printf("        table       -> Fully qualified target table name.\n\n");
  printf("Will prompt for optional index generation.\n\n");
  printf("Error Messages in DBMIMPORT.MSG.\n\n");

  return;

}  /* end helpmsg */
```

# DBLIST.SQC

## List System or Volume Directory

```
/************************************************************************/
/*  PROGRAM NAME: dblist.sqc                                            */
/*                                                                      */
/*  PURPOSE: List system or volume directory.                           */
/*                                                                      */
/*  FORMAT: dblist [drive]                                              */
/*                                                                      */
/*  INPUT:   [drive] = D    -> list system database directory           */
/*           [drive] = A-Z  -> list volume directory for specified      */
/*                             file system                              */
/*           [drive] = ?    -> returns help information                 */
/*                                                                      */
/*  OUTPUT:  Listing of databases for whole system                      */
/*                      OR                                              */
/*           Listing of databases on particular file system             */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary                                              */
/*                                                                      */
/*  NOTES:   Database Manager does not have to be started for this      */
/*           program to run.                                            */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                  constant definitions                                */
/************************************************************************/

#include <stdio.h>
#include <string.h>

#include <sqlenv.h>                     /* SQL Environment include     */
#include <sql.h>                        /* SQL constants & err msg fn  */

/************************************************************************/
/*                  Global data elements                                */
/************************************************************************/

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area     */

/************************************************************************/
/*                  forward declare of routines                         */
/************************************************************************/

void errmsg(void);                      /* error message routine       */
void helpmsg(void);                     /* print out user help         */
```

# DBLIST.SQC (CONT'D)

```
/*****************************************************************************/
/*                        M A I N   R O U T I N E                            */
 *****************************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{

struct sqledinfo *myinfo;
int rc;

unsigned short handle, entries, i;
unsigned char drive = 'D';
#define LINE_SZ 76
unsigned char tbuf[LINE_SZ];

  if (argc >= 2 && *argv[1] == '?')
     {
     helpmsg();                          /*   print out help           */
     return(0);                          /*   leave program            */
     }

  if (argc > 1)                        /* if parameters specified  . . */
     drive = *argv[1];                 /*      use specified drive     */
/*   open directory for scanning                                        */
  rc = sqledops(drive,&handle,&entries,&sqlca);    /* open directory  */
  if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)       /* error???        */
      errmsg();                                    /* yes - report it */
  else                                             /* entries found   */
    {

    /* print out header                                                */

    printf(" Dr   Name    Alias    Comment             Directory Type   Node\n");
    printf(" --   ----    -----    -------             --------- ----   ----\n");

    memset(tbuf,' ',LINE_SZ);                 /* init buffer          */

    /*   print out each directory entry                                */

    for (i=1; i<=entries; i++)
      {

      sqledgne(handle,&myinfo,&sqlca);        /* get directory entry  */

      strncpy(&tbuf[1],myinfo->drive,SQL_DRIVE_SZ);
      strncpy(&tbuf[4],myinfo->dbname,SQL_DBNAME_SZ);
      strncpy(&tbuf[13],myinfo->alias,SQL_ALIAS_SZ);
      strncpy(&tbuf[22],myinfo->comment,SQL_CMT_SZ);
      strncpy(&tbuf[53],myinfo->intname,SQL_INAME_SZ);
      strncpy(&tbuf[68],myinfo->nodename,SQL_NNAME_SZ);

      if (myinfo->type == SQL_HOME)
         strncpy(&tbuf[63],"HOME",4);
      else if (myinfo->type == SQL_INDIRECT)
         strncpy(&tbuf[63],"IND ",4);

      printf("%s\n",tbuf);                    /* print formatted entry*/

     }  /* end for */
```

# DBLIST.SQC (CONT'D)

```
    sqledcls(handle,&sqlca);                    /* close directory scan */

    }  /* end - entries found */

  return(0);

}  /* end main */
/*****************************************************************
/*                          E R R M S G                          */
/*****************************************************************/
void errmsg()
{

#define BUFSIZE 512#
define WIDTH 75
unsigned char buffer[BUFSIZE];

  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);          /* get error msg text*/

  printf("%s\n",buffer);                          /* print msg text    */

  return;

}   /* end errmsg */

/*****************************************************************
/*                          H E L P M S G                        *
/*****************************************************************

void helpmsg()
{

  printf("DBLIST will display your database directory.\n\n");
  printf("FORMAT: DBLIST [drive]\n\n");
  printf("WHERE:  [drive] = 0    -> list system database directory\n");
  printf("        [drive] = A-Z  -> list volume directory for specified\n");
  printf("                          file system\n");

  return;

}  /* end helpmsg */
```

# DBSTRUCT.SQC

## List Tables and Views in Requested Database

```
/************************************************************************/
/*  PROGRAM NAME: dbstruct.sqc                                          */
/*                                                                      */
/*  PURPOSE: List all tables and views in requested database.           */
/*           Also list the columns in each table/view.                  */
/*                                                                      */
/*  FORMAT: dbstruct dbname [option]                                    */
/*                                                                      */
/*  INPUT:   dbname          -> database name (up to eight characters)  */
/*                                                                      */
/*  PROMPTS FOR:                                                        */
/*                                                                      */
/*           [password]      -> optional password (8 char max)          */
/*                                                                      */
/*  OUTPUT:  a formatted report of tables (and their columns).          */
/*           Output is sent to standard output and may be redirected    */
/*           to a file or printer.                                      */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary, automatic when run                          */
/*                                                                      */
/*  NOTES:   Database Manager must be started before execution          */
/*           of this program.                                           */
/*                                                                      */
 ************************************************************************/

 ************************************************************************/
/*                 constant definitions                                 */
/************************************************************************/

#include <stdio.h>
#include <string.h>

#include <sql.h>
#include <sqlenv.h>
#include <sqlcodes.h>

EXEC SQL INCLUDE sqlca;

/************************************************************************/
/*                 Global data elements                                 */
/************************************************************************/

#define BUFSIZE 512
#define WIDTH 75
#define NOTFOUND SQL_RC_W100

unsigned char *step;                /* pointer to program step text   */
unsigned char password[9]="";       /* database password              */
unsigned char msgbuf[BUFSIZE];      /* place for returned error msg   */
unsigned char saved_tname[19];      /* current table being processed  */
```

# DBSTRUCT.SQC (CONT'D)

```
/*****************************************************************************
/*              Error Handling Options For SQL Statements                    */
/*****************************************************************************

EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL WHENEVER SQLWARNING CONTINUE;

EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*****************************************************************************/
/*                    SQL Declaration Section                                */
/*****************************************************************************/

EXEC SQL BEGIN DECLARE SECTION;

unsigned char tcreator[9];          /* SYSTABLES.CREATOR  CHAR(8)       */

unsigned char tname[19];            /* SYSTABLES.NAME     VARCHAR(18)   */

unsigned char tcomment[255];        /* SYSTABLES.REMARKS  VARCHAR(254)  */
short         tcomment_nid;

unsigned char cname[19];            /* SYSCOLUMNS.NAME    VARCHAR(18)   */

unsigned char ccomment[255];        /* SYSCOLUMNS.REMARKS VARCHAR(254)  */
short         ccomment_nid;

unsigned char ctype[9];             /* SYSCOLUMNS.COLTYPE CHAR(8)       */

short         clength;              /* SYSCOLUMNS.LENGTH  SMALLINT      */

unsigned char cnull[2];             /* SYSCOLUMNS.NULLS   CHAR(1)       */

short         colno;                /* SYSCOLUMNS.COLNO   SMALLINT      */

EXEC SQL END DECLARE SECTION;

/*****************************************************************************
/*                  forward declare of routines                              */
/*****************************************************************************

void helpmsg(void);                         /* print out user help          */

/*****************************************************************************/
/*                            MAIN ROUTINE                                   */
/*****************************************************************************/

int main(argc, argv)
int argc;
har *argv[];
{

  int errorlvl = 0;
  int rc = 0;

  if ( (argc >= 2 && *argv[1] == '?') || (argc < 2) )

     {
     helpmsg();                             /*   print out help             */
     return(0);                             /*   leave program              */
     }
```

# DBSTRUCT.SQC (CONT'D)

```
  /*  Declare Cursor                                                    */

  EXEC SQL DECLARE current_row CURSOR FOR
           SELECT T.CREATOR, T.NAME, T.REMARKS,
                  C.NAME, C.REMARKS, C.COLTYPE, C.LENGTH, C.NULLS, C.COLNO
                  FROM SYSIBM.SYSTABLES T, SYSIBM.SYSCOLUMNS C
                  WHERE T.NAME=C.TBNAME
                  ORDER BY T.CREATOR, T.NAME, C.COLNO;

sqleisig(&sqlca);              /* Install Signal Handler              */

 ********************************************************************/
/*  Start Using the Database                                         */
/********************************************************************/

step = "Start Using Database";
sqlestrd(argv[1],password,SQL_USE_SHR,&sqlca);

/* if a password exists (assumed null password failed) ...           */

if (sqlca.sqlcode == SQLE_RC_INVPSW)
   {
   printf("Enter database password: ");          /* ask for password*/
   gets(password);

   sqlestrd(argv[1],password,SQL_USE_SHR,&sqlca);/* try again        */
   }

/* if a restart required ...                                         */

if (sqlca.sqlcode == SQLE_RC_DB_RESTART)
   {
    printf("Database Restart in progress. Please wait . . .\n");

    step = "Restart Database";
    sqlerest(argv[1],password,&sqlca);
   }

if (sqlca.sqlcode != SQL_RC_OK)         /* if sql error              */
   goto exit_error;                     /* go to error handling      */


/********************************************************************/
/*  Open the Cursor, Handle any (re)binding                          */
/********************************************************************/

EXEC SQL OPEN current_row;

/* if this application is not bound to the requested database ...    */

if (sqlca.sqlcode == SQL_RC_E818 || sqlca.sqlcode == SQL_RC_E805)

   {
    step = "Bind Application to Database";
    sqlabind("dbstruct.bnd",argv[1],password,"",NULL,&sqlca);

    if (sqlca.sqlcode != SQL_RC_OK)   /* error on rebind             */
       goto exit_error;
    else
       EXEC SQL OPEN current_row;     /* try to reopen the cursor    */
   }
```

```
if (sqlca.sqlcode != SQL_RC_OK)       /* if error on opening cursor */
   goto exit_error;


/*****************************************************************
/*  Start Retrieving Rows From the Result                        */
/*****************************************************************

EXEC SQL WHENEVER SQLERROR GOTO exit_error;

EXEC SQL FETCH current_row into :tcreator,
                                :tname,
                                :tcomment :tcomment_nid,
                                :cname,
                                :ccomment :ccomment_nid,
                                :ctype,
                                :clength,
                                :cnull,
                                :colno;

strcpy(saved_tname,tname);

while (sqlca.sqlcode != NOTFOUND)
      {

       printf("\nOwner        .Table Name                    Comment");
       printf("\n-----        -----------                    -------");
       printf("\n%s.%-18s [%s]\n",tcreator,tname,tcomment);
       printf("\n         Column Name      Type    Length Null Comment");
       printf("\n         -----------      ----    ------ ---- -----\n");

       do
        {

         printf("         %-18s %s %6d %-4s [%s]\n",
                cname,ctype,clength,cnull,ccomment);

         EXEC SQL FETCH current_row into :tcreator,
                                         :tname,
                                         :tcomment :tcomment_nid,
                                         :cname,
                                         :ccomment :ccomment_nid,
                                         :ctype,
                                         :clength,
                                         :cnull,
                                         :colno;

        } while ((sqlca.sqlcode != NOTFOUND) &&
          (strcmp(tname,saved_tname)==0));

       strcpy(saved_tname,tname);

      }  /* end outer loop for each table */

EXEC SQL CLOSE current_row;

EXEC SQL COMMIT WORK;
```

# DBSTRUCT.SQC (CONT'D)

```
/*****************************************************************/
/*  Normal program exit                                          */
/*****************************************************************/

exit:
  sqlestpd(&sqlca);

  return(errorlvl);

/*****************************************************************/
/*           E N D   O F   M A I N   P R O G R A M               */
/*****************************************************************/

/*****************************************************************/

/*****************************************************************/
/*                  ABNORMAL EXIT                                */
/*****************************************************************/

exit_error:

  printf("\nSQL error %d during \"%s\"\n",sqlca.sqlcode,step);

  rc = sqlaintp(msgbuf,BUFSIZE,WIDTH,&sqlca);

  if (rc < 0)
     printf("\nSQLAINTP ERROR. Return code = %d\n",rc);

  if (rc > 0)
     printf("\n%s",msgbuf);

  EXEC SQL WHENEVER SQLERROR CONTINUE;

  EXEC SQL ROLLBACK WORK;

  if (sqlca.sqlcode != 0)
     printf("\nSQLCODE is %ld, ROLLBACK failed.",sqlca.sqlcode);

  errorlvl=2;

  goto exit;
}

/*****************************************************************/
/*                          H E L P M S G                        */
/*****************************************************************/

void helpmsg()
{

  printf("DBSTRUCT diplays the tables (and columns) contained in the\n");
  printf("         requested database.\n\n");
  printf("FORMAT: dbstruct database\n\n");
  printf("WHERE:  database     -> name of database to list structure for.\n\n");

  return;

}  /* end helpmsg */
```

# ERASEDB.SQC

## Erase the Named Database

```
/************************************************************************/
/*  PROGRAM NAME: erasedb.sqc                                           */
/*                                                                      */
/*  PURPOSE: Erase the named database                                   */
/*                                                                      */
/*  FORMAT:  erasedb dbname                                             */
/*                                                                      */
/*  INPUT:   dbname     -> reference name for database (8 char max)     */
/*                                                                      */
/*  PROMPTS FOR:                                                        */
/*                                                                      */
/*           [password]     -> optional password (8 char max)           */
/*                                                                      */
/*  OUTPUT:  specified database erased                                  */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary                                              */
/*                                                                      */
/*  NOTES:   Database Manager does not have to be started for this      */
/*           program to run.                                            */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                 constant definitions                                 */
/************************************************************************

#include <stdio.h>
#include <string.h>

#include <sqlenv.h>                     /* SQL Environment include      */
#include <sql.h>                        /* SQL constants & err msg fn   */

/************************************************************************/
/*                 Global data elements                                 */
/************************************************************************

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area      */
unsigned char password[9] = "\0\0\0\0\0\0\0\0";

/************************************************************************/
/*                 forward declare of routines                          */
/************************************************************************/

void errmsg(void);                      /* error message routine        */
void helpmsg(void);                     /* print out user help          */

/************************************************************************/
/*                   M A I N   R O U T I N E                            */
/************************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{
```

# ERASEDB.SQC (CONT'D)

```
int rc;

  if ( (argc >= 2 && *argv[1] == '?') || (argc < 2) )
     {
     helpmsg();                          /*   print out help          */
     return(0);                          /*   leave program           */
     }

  printf("Enter database password; (press 'Enter' for none): ");
  gets(password);

  rc = sqledrpd(argv[1],password,&sqlca);

  if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)       /* error???        */

      errmsg();                                    /* yes - report it */

  else

      printf("Done!\n");

  return(rc);

}  /* end main */

/*****************************************************************************/
/*                          E R R M S G                                      */
/*****************************************************************************/
void errmsg()
{

#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];


  splaintp(buffer,BUFSIZE,WIDTH,&splca);  /* get error msg text      /

  printf("%s\n",buffer);                  /* print msg text          */

  return;

}   /* end errmsg */

/*****************************************************************************/
/*                          H E L P M S G                                    */
/*****************************************************************************/
void helpmsg()
{

  printf("ERASEDB removes the specified database from your system.\n\n");
  printf("FORMAT: ERASEDB dbname\n\n");
  printf("WHERE:  dbname    -> reference name for database to be erased.\n\n");
  printf("PROMPTS FOR: password\n");

  return;

}  /* end helpmsg */
```

# FILLTB.SQC

## Add Data Rows to Table Created by INSTALL

```
/*****************************************************************************/
/*  PROGRAM NAME: filltb.sqc                                                 */
/*                                                                           */
/*  PURPOSE: Add data rows to the table created by INSTALL                   */
/*                                                                           */
/*  DESCRIPTION: Contains subroutines used by INSTALL                        */
/*                                                                           */
/*  PROCEDURE NAME: filltable                                                */
/*                                                                           */
/*  FORMAT: filltable()                                                      */
/*                                                                           */
/*  INPUT:  argv[1] from CREATTB, containing database name                   */
/*                                                                           */
/*          password from INSTALL containing db password                     */
/*                                                                           */
/*  OUTPUT: Data rows added to the staff table created by INSTALL            */
/*                                                                           */
/*  PRECOMPILE: sqlprep filltb.sqc dbname /b                                 */
/*                                                                           */
/*              - where dbname is the name of a database containing          */
/*                the staff table created in INSTALL. This table             */
/*                must exist so the precompiler can check the                */
/*                INSERT statements contained here.                          */
/*                                                                           */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                        */
/*                                                                           */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa                    */
/*                                                                           */
/*  BIND:    Not necessary, automatic when run                               */
/*                                                                           */
/*  NOTES:   Database Manager must be started before execution               */
/*           of this program.                                                */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/*                 constant definitions                                      */
/*****************************************************************************/

#include <stdio.h>
#include <string.h>

#include <sql.h>
#include <sqlenv.h>
#include <sqlcodes.h>

#include <sqlca.h>                      /* definition of sqlca               */
extern struct sqlca sqlca;              /* sqlca declared in INSTALL         */

/*****************************************************************************/
/*            Error Handling Options For SQL Statements                      */
/*****************************************************************************/

EXEC SQL WHENEVER SQLERROR GOTO exit;

EXEC SQL WHENEVER SQLWARNING CONTINUE;

EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

# FILLTB.SQC (CONT'D)

```
/*****************************************************************************/
/*                          FILLTABLE                                        */
/*****************************************************************************/

long filltable(db,password)

unsigned char *db;
unsigned char *password;

{

/*****************************************************************************/
/*  Bind this program to the new database                                    */
/*****************************************************************************/

sqlabind("filltb.bnd",db,password,"",NULL,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)       /* error on rebind            */
   goto exit;


/*****************************************************************************/
/*  Insert data into the table                                               */
/*****************************************************************************/

EXEC SQL INSERT INTO staff ( ID, NAME, DEPT, JOB, YEARS, SALARY)
                    VALUES ( 10, 'Sanders', 20, 'Mgr', 5, 18357.50);

EXEC SQL INSERT INTO staff ( ID, NAME, DEPT, JOB, YEARS, SALARY, COMM)
                   VALUES ( 20, 'Pernal', 20, 'Sales', 8, 18171.25, 612.45);

EXEC SQL COMMIT WORK;


exit:
  return(sqlca.sqlcode);

}
```

# INSTALL.SQC

## Simulate Application Installation Process

```
/************************************************************************/
/*  PROGRAM NAME: install.sqc                                           */
/*                                                                      */
/*  PURPOSE: Simulate an application installation process.              */
/*           Create a sample database with one table. Creates some      */
/*           rows in the sample table.                                  */
/*                                                                      */
/*  FORMAT: install drive dbname                                        */
/*                                                                      */
/*  INPUT:   drive          -> file system on which to create db        */
/*                                                                      */
/*           dbname         -> database name (up to eight characters)   */
/*                                                                      */
/*                                                                      */
/*  PROMPTS FOR:                                                        */
/*                                                                      */
/*           [password]     -> optional password (8 char max)           */
/*                                                                      */
/*           [comment]      -> optional comment (30 char max)           */
/*                                                                      */
/*  OUTPUT:  Database <dbname> with sample table                        */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary, automatic when run                          */
/*                                                                      */
/*  NOTES:   Database Manager must be started before execution          */
/*           of this program.                                           */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                 constant definitions                                 */
/************************************************************************/

#include <stdio.h>
#include <string.h>

#include <sql.h>
#include <sqlenv.h>
#include <sqlcodes.h>

EXEC SQL INCLUDE sqlca;

/************************************************************************/
/*                 Global data elements                                 */
/************************************************************************/

#define BUFSIZE 512
#define WIDTH 75
#define NOTFOUND SQL_RC_W100
```

# INSTALL.SQC (CONT'D)

```
unsigned char *step;                /* pointer to program step text   */
unsigned char password[9]="";       /* database password              */
unsigned char comment[31] = "";     /* database comment               */
unsigned char msgbuf[BUFSIZE];      /* place for returned error msg   */
unsigned char saved_tname[19];      /* current table being processed  */

/*****************************************************************/
/*            Error Handling Options For SQL Statements          */
/*****************************************************************/

EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL WHENEVER SQLWARNING CONTINUE;

EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*****************************************************************/
/*                forward declare of routines                    */
/*****************************************************************/

extern long filltable( unsigned char *, unsigned char *);
void helpmsg(void);                     /* print out user help        */

/*****************************************************************/
/*                         MAIN ROUTINE                          */
/*****************************************************************/

int main(argc, argv)
int argc;
char *argv[];
{

  int errorlvl = 0;
  long rc = 0;

  if ( (argc > 1 && *argv[1] == '?') || (argc < 3) )  /* help request */
     {
     helpmsg();                         /*   print out help           */
     return(0);                         /*   leave program            */
     }

  sqleisig(&sqlca);             /* Install Signal Handler             */

/*****************************************************************/
/*  Create the requested database                                */
/*****************************************************************/


printf("Enter database password; (press 'Enter' for none): ");
gets(password);

printf("Enter database comment: ");
gets(comment);

step = "Create Database";
rc = sqlecred(argv[2],*argv[1],password,comment,0,&sqlca);

if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)       /* error???        */
   goto exit_error;                              /* yes - report it */
```

# INSTALL.SQC (CONT'D)

```
/*****************************************************************/
/*  Start Using the Database                                     */
/*****************************************************************/

step = "Start Using Database";
rc = sqlestrd(argv[2],password,SQL_USE_SHR,&sqlca);

if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)  /* if sql error         */
   goto exit_error;                         /* go to error handling */

/*****************************************************************/
/*  Bind this program to the new database                        */
/*****************************************************************/

step = "Bind Application to Database";
sqlabind("install.bnd",argv[2],password,"",NULL,&sqlca);

if (sqlca.sqlcode != SQL_RC_OK)       /* error on rebind          */
   goto exit_error;

/*****************************************************************
/*  Create the sample table                                      */
/*****************************************************************

EXEC SQL CREATE TABLE staff (ID     SMALLINT NOT NULL,
                             NAME   VARCHAR(9),
                             DEPT   SMALLINT,
                             JOB    CHAR(5),     /* */
                             YEARS  SMALLINT,
                             SALARY DECIMAL(7,2),
                             COMM   DECIMAL(7,2));

/*****************************************************************/
/*  Insert data into the table                                   */
/*****************************************************************/

rc = filltable(argv[2],password);     /* insert the data rows     */

if (rc != SQL_RC_OK)                  /* check for SQL errors     */
   goto exit_error;

/*****************************************************************
/*  Normal program exit                                          */
/*****************************************************************/

exit:
sqlestpd(&sqlca);                     /* stop using database      *

return(errorlvl);

/*****************************************************************/
/*            E N D   O F   M A I N   P R O G R A M              */
/*****************************************************************/

/*****************************************************************/
```

# INSTALL.SQC (CONT'D)

```
/*****************************************************************************/
/*                  ABNORMAL EXIT                                            */
/*****************************************************************************/
exit_error:

  printf("\nSQL error %d during \"%s\"\n",sqlca.sqlcode,step);

  rc = sqlaintp(msgbuf,BUFSIZE,WIDTH,&sqlca);

  if (rc < 0)
     printf("\nSQLAINTP ERROR. Return code = %d\n",rc);

  if (rc > 0)
     printf("\n%s",msgbuf);

  EXEC SQL WHENEVER SQLERROR CONTINUE;

  EXEC SQL ROLLBACK WORK;

  if (sqlca.sqlcode != 0)
     printf("\nSQLCODE is %ld, ROLLBACK failed.",sqlca.sqlcode);

  errorlvl=2;

  goto exit;
}

 *****************************************************************************/
/*                          H E L P M S G                                    */
 *****************************************************************************/
void helpmsg()
{

  printf("INSTALL creates the requested database and build a sample\n");
  printf("         table, STAFF.\n\n");
  printf("FORMAT:  INSTALL drive database\n\n");
  printf("WHERE:   drive  = A-Z  -> file system to contain the database\n");
  printf("         drive  = \"\"   -> create database on file system where\n");
  printf("                           the Database Manager was installed\n");
  printf("         database      -> name for the new database\n\n");
  printf("PROMPTS FOR: optional comment and password\n");

  return;

}  /* end helpmsg */
```

# LOGFILE.SQC

## Adjust Logfile Configuration Parameters

```
/************************************************************************/
/*  PROGRAM NAME: logfile.sqc                                           */
/*                                                                      */
/*  PURPOSE: Adjust the logfile configuration parameters.               */
/*                                                                      */
/*  FORMAT: logfile [database]                                          */
/*                                                                      */
/*  INPUT:   [database]     -> name of database (contains logfile)      */
/*                                                                      */
/*  OUTPUT: logfile parameters changed                                  */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa+llibcdll      */
/*                                                                      */
/*  BIND:    Not necessary, automatic when executed                     */
/*                                                                      */
/*  NOTES:   Database Manager does not have to be started for this      */
/*           program to run.                                            */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                  constant definitions                                */
/************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <sqlutil.h>                    /* SQL utility functions      */
#include <sqlenv.h>                     /* SQL environment fns        */
#include <sql.h>                        /* SQL constants & err msg fn */

/************************************************************************/
/*                  Global data elements                                */
/************************************************************************/

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area    */

unsigned char password[9]="";       /* database password              */

unsigned char keep_at_it[1]="";

unsigned char database[9];                  /* database name          */
#define LOGFILE_PARMS 6                     /* # of logfile parms     */
struct sqlfupd logfile_array[LOGFILE_PARMS]; /* token/value array     */

unsigned long minlog=0;
unsigned long maxlog=0;
unsigned long logext=0;
unsigned long country;
unsigned long codepage;
```

# LOGFILE.SQC (CONT'D)

```
struct majorminor
{
  unsigned char minor;
  unsigned char major;
} release;

unsigned char minlog_text[4];       /* text for new value             */
unsigned char maxlog_text[4];       /* text for new value             */
unsigned char logext_text[4];       /* text for new value             */

#define BUFSIZE 512
#define WIDTH 75

/*****************************************************************************/
/*                 forward declare of routines                               */
/*****************************************************************************/

void display_settings(void);            /* display logfile parms      */
void errmsg(void);                      /* error message routine      */
void helpmsg(void);                     /* print out user help        */

/*****************************************************************************/
/*                          MAIN ROUTINE                                     */
 *****************************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{

  if(argc >= 2)                             /* argument 1 exists        */
    strcpy(database,argv[1]);
  else
    {
    printf("Enter Database Name: ");
    gets(database);
    }

 /*****************************************************************************/
/*  Initialize Logfile parameter array                                       */
 *****************************************************************************/

logfile_array[0].token    = SQLF_DBTN_LOGFILE;
logfile_array[0].ptrvalue = (char *)&minlog;

logfile_array[1].token    = SQLF_DBTN_LOGEXT;
logfile_array[1].ptrvalue = (char *)&logext;

logfile_array[2].token    = SQLF_DBTN_LOGMAXEXT;
logfile_array[2].ptrvalue = (char *)&maxlog;

logfile_array[3].token    = SQLF_DBTN_COUNTRY;
logfile_array[3].ptrvalue = (char *)&country;

logfile_array[4].token    = SQLF_DBTN_CODEPAGE;
logfile_array[4].ptrvalue = (char *)&codepage;

logfile_array[5].token    = SQLF_DBTN_RELEASE;
logfile_array[5].ptrvalue = (char *)&release;
```

# LOGFILE.SQC (CONT'D)

```
display_settings();

while (keep_at_it[0] != 0)
  {

  printf("\nMinimum logfile in 4K pages (12 to 2**32-1): ");
  gets(minlog_text);
  minlog = atol(minlog_text);

  printf("Each extension in 4K pages (4 to 256): ");
  gets(logext_text);
  logext = atol(logext_text);

  printf("Maximum allowed extensions (0 to 2,000,000): ");
  gets(maxlog_text);
  maxlog = atol(maxlog_text);

  sqlfeudb(database,password,3,logfile_array,&sqlca);

  if (sqlca.sqlcode != SQL_RC_OK)                /* error???        */
        errmsg();                                /* yes - report it */

  display_settings();

  }

return(0);

}  /* end main */

/*****************************************************************
/*           E N D   O F   M A I N   P R O G R A M               */
/*****************************************************************

/*****************************************************************/
/*              D I S P L A Y _ S E T T I N G S                  */
/*****************************************************************/

void display_settings()
{
long temp;

  sqlfxdbc(database,password,LOGFILE_PARMS,logfile_array,&sqlca);

  if (sqlca.sqlcode != SQL_RC_OK)                  /* error???        */
        errmsg();                                  /* yes - report it */

  printf("\nDatabase...... %s",database);
  printf("\nCountry Code.. %ld",country);
  printf("\nCodepage...... %ld",codepage);
  printf("\nDBM Release... %u.%u\n\n",release.major,release.minor);

  printf("The current logfile parameters are:\n\n");
  printf("Minimum size..... %-9ld 4K blocks = %9ld Bytes\n",
        minlog,minlog*4096);
  printf("Each extension..... %-9ld 4K blocks = %9ld Bytes\n",
        logext,logext*4096);
  printf("Maximum extensions... %-9ld           = %9ld Bytes\n",
        maxlog,maxlog*logext*4096);
  printf("Largest logfile..... %-9ld Bytes\n",(minlog+(maxlog*logext))*4096);
```

# LOGFILE.SQC (CONT'D)

```
  printf("\nDo you want to reset these? (blank to exit): ");
  gets(keep_at_it);

  return; /* end display_settings */
}

/*****************************************************************************/
/*                          E R R M S G                                      */
/*****************************************************************************/

void errmsg(){

#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];

  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);  /* get error msg text      */

  printf("\n%s\n",buffer);                /* print msg text          */

  exit(1);

}   /* end errmsg */

/*****************************************************************************/
/*                          H E L P M S G                                    */
/*****************************************************************************/
void helpmsg()
{

  printf("LOGFILE allows you to change a database's logfile parameters.\n\n");
  printf("FORMAT: logfile [database]\n\n");
  printf("WHERE:  [database]     -> name of the database containing
         the logfile.\n");

  return;
```

# QUERY.SQC

## Dynamic SQL Programming Example

```
/*****************************************************************************
/*  PROGRAM NAME:        QUERY.SQC                                           */
/*                                                                           */
/*  PURPOSE:             Dynamic SQL Programming Example                     */
/*                                                                           */
/*  PROGRAM LANGUAGE:    'C' Language                                        */
/*                                                                           */
/*  COPYRIGHT:           (C) COPYRIGHT IBM CORP. 1988                        */
/*                       LICENSED MATERIAL - PROGRAM PROPERTY OF IBM         */
/*                       ALL RIGHTS RESERVED                                 */
/*                                                                           */
/*  TRADEMARK:           OS/2 and OS/2 EE are trademarks of IBM              */
/*                                                                           */
/*  PROGRAM DESCRIPTION:                                                     */
/*                       This sample program uses dynamic SQL to execute     */
/*                       SQL statements against a user-specified database.   */
/*                       The SQL statements are entered after a prompt.      */
/*                       For SELECT statements, the resulting data is        */
/*                       displayed on the screen in tabular format           */
/*                                                                           */
/*                       The program is initiated by typing the following    */
/*                       command at the OS/2 command prompt:                 */
/*                           query                                           */
/*                                                                           */
/*                       After initiation the user is prompted for a         */
/*                       database name, and a database password (if needed). */
/*                       The program performs a 'Start Using Database'       */
/*                       against the specified database.                     */
/*                                                                           */
/*                       The program goes into a loop, prompting for an      */
/*                       SQL statement and executing that statement.         */
/*                                                                           */
/*                       The program terminates when a null Enter is         */
/*                       received in response to the prompt for an           */
/*                       SQL statement.                                      */
/*                                                                           */
/*  INPUT:               Upon invocation, the user is prompted for:          */
/*                       - database name = database to be connected to       */
/*                       - password = database password, if one exists,      */
/*                         else press Enter.                                 */
/*                                                                           */
/*                       The program then repeatedly prompts for:            */
/*                       - statement string = SQL statement                  */
/*                                                                           */
/*  OUTPUT:              For SELECT statements, the result of the SELECT     */
/*                       is displayed on the screen, with each row of        */
/*                       output preceded by a line number.                   */
/*                                                                           */
/*                       For all other statements, there is no visual        */
/*                       response.  The program simply performs the          */
/*                       requested function and prompts for the next         */
/*                       statement.                                          */
/*                                                                           */
/*  PRECOMPILE: any database, delay bind                                     */
/*                                                                           */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                        *
```

# QUERY.SQC (CONT'D)

```
/*                                                                          */
/*  LINK:     /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa                  */
/*                                                                          */
/*  BIND:     Not necessary, automatic when run                             */
/*                                                                          */
/****************************************************************************/

 /***************************************************************************/
/*                          H E L P M S G                                  */
/***************************************************************************/
void helpmsg()
{

  printf("LOGFILE allows you to change a database's logfile parameters.\n\n");
  printf("FORMAT: logfile [database]\n\n");
  printf("WHERE:  [database]     -> name of the database containing
     the logfile.\n");

  return;

}  /* end helpmsg */

 /***************************************************************************/
/*  INCLUDE FILES                                                          */
/***************************************************************************/

#include <malloc.h>             /* Include files for C built-in functions    */
#include <memory.h>
#include <stdio.h>
#include <string.h>

#include <doscalls.h>           /* Include file for OS/2 function calls      */

#include "sql.h"                /* Database Manager System constants         */
#include "sqlcodes.h"           /* SQL return code constants                 */
#include "sqlenv.h"             /* Database Environment Command interfaces   */

/***************************************************************************/
/*  EXTERNAL DECLARATIONS                                                  */
/***************************************************************************/

#define PRIVATE   0             /* Used on call to DOSALLOCSEG               */

EXEC SQL INCLUDE SQLCA;         /* Define an sqlca for SQL return codes      */
EXEC SQL INCLUDE SQLDA;         /* Include declaration of sqlda structure    */

EXEC SQL BEGIN DECLARE SECTION; /* Host variable declare section             */
  char prep_string[2000];       /* String to hold dynamic SQL statements     */
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE c1 CURSOR FOR s1;      /* Declare a cursor to be used with  */
                                        /*  PREPAREd SELECT statements.      */

int rc = 0;                               /* Initialize return code to 0     */
char dbname[20];                          /* Database name to connect to     */
char getpswd[9];                          /* String to retrieve password     */
char password[9] = "\0\0\0\0\0\0\0\0";  /* Database password; init to null */

static short sqlsize = 0;                 /* Number of sqlvars in sqlda      */
```

# QUERY.SQC (CONT'D)

```
static union {                                  /* Pointer to FAR char string      */
               struct
               {
                 unsigned offset;
                 unsigned selector;
               } ptr4;
               char far * ptr;
             } segptr;

static struct sqlda far *sqlda = NULL;  /* pointer to minimum SQLDA          *

static short valid_vars = 0;            /* Number of items in a select list  *
unsigned short adjust;                  /* Adds two bytes to storage alloc'd
                                                for varchar and long varchar *

#define BUFSIZE 512                     /* Buffer size for error messages    *
unsigned char msgbuf[BUFSIZE];          /* Message buffer                    *

#define LONG_DISP_LEN 4000;             /* String Length used to display

/*****************************************************************************/
/*                  forward declare of routines                              */
/*****************************************************************************/

int main(int, char *[]);
static int  fetch(void);

/*****************************************************************************/
/*                           MAIN ROUTINE                                    */
/*****************************************************************************/

main(argc, argv)
int argc;                       /* number of pointers to input parameters    *
char *argv[];                   /* array of pointers to input parameters     *

{
   int i;
   int rc = 0;

/*****************************************************************************/
/*  Start Using the Database                                                 */
/*****************************************************************************/

 printf("Enter database name: ");
 gets(dbname);

 sqlestrd(dbname,password,SQL_USE_SHR,&sqlca);   /* try null pw      */

/* if a password exists (assumed null password failed) ...          */

 if (sqlca.sqlcode == SQLE_RC_INVPSW)
    {
    printf("Enter database password: ");          /* ask for password*/
    gets(password);

    sqlestrd(dbname,password,SQL_USE_SHR,&sqlca); /* try again        */
    }
```

# QUERY.SQC (CONT'D)

```
/* if any other error . . .                                          */

 if (sqlca.sqlcode != 0)
   {
      printf("Start DB Error:  sqlcode = %ld.\n", sqlca.sqlcode);

      rc = sqlaintp(msgbuf,BUFSIZE,0,&sqlca);

      if (rc < 0)               /* message retrieve err*/
      {
          printf ("SQLAINTP ERROR. Return code = %d\n",rc);
      }

      if (rc > 0)                /* error message return*/
      {
          printf ("%s\n",msgbuf);
      }
   }

 else

/*********************************************************************/
/*  Successful Start - Allocate Minimum SQLDA, Initialize Header     */
 *********************************************************************/

   {

   sqlda = malloc(SQLDASIZE(0));
   memset((char *)sqlda, '\0',SQLDASIZE(0));
   sqlda->sqln = 0;
   sqlda->sqld = 0;

 /* Prompt for the first SQL statement                               */

   printf("\nSQL stmt: ");
   gets(prep_string);

 *********************************************************************/
/*  Loop until no more input . .                                     */
/*********************************************************************/

   while (prep_string[0] != 0)
     {

   /* Reset the return code 'rc'.                                    */
   /* Free the storage obtained for retrieving values from           */
   /* the last SELECT statement, and reset 'valid_vars'.             */
   /* ('valid_vars' is the number items in the select list.)         */

     rc = 0;
     for (i = 0; i<valid_vars; i++)
       {
       free(sqlda->sqlvar[i].sqldata);  /* release variable     */
       free(sqlda->sqlvar[i].sqlind);   /* release null ind     */
       }
     valid_vars = 0;
```

# QUERY.SQC (CONT'D)

```
   /*****************************************************************/
   /*  PREPARE the SQL statement into the minimum SQLDA             */
   /*****************************************************************/

     EXEC SQL PREPARE s1 INTO :*sqlda FROM :prep_string;

   /* if this application is not bound to the requested database ...*/

     if (sqlca.sqlcode == SQL_RC_E818 || sqlca.sqlcode == SQL_RC_E805)

         {
         sqlabind("query.bnd",dbname,password,"",NULL,&sqlca);

         if (sqlca.sqlcode >= 0)

           EXEC SQL PREPARE s1 INTO :*sqlda FROM :prep_string;
         }

   /* If the sqlca.sqlcode field is OK and the statement is a       */
   /* SELECT statement, then set up the sqlda for retrieving        */
   /* values. (Note: the sqlda->sqld field tells the number of      */
   /* items in the select list, hence if it is greater than 0 the   */
   /* statement is a SELECT statement.)                             */

     sqlsize = sqlda->sqld;
     if ((sqlca.sqlcode >= 0) && (sqlsize > 0))
       {

/*****************************************************************
/*  SELECT STATEMENT - Reallocate SQLDA and DESCRIBE SELECT   */
/*****************************************************************

 sqlda = realloc(sqlda, SQLDASIZE(sqlsize));
 memset((char *)sqlda, '\0',SQLDASIZE(sqlsize));
 sqlda->sqln = sqlsize;

 EXEC SQL DESCRIBE s1 INTO :*sqlda;

/* For each sqlvar element in the sqlda, allocate any memory  */
/* required by the sqldata and sqlind pointers.               */
/* (Note:  the data type and length fields are set up by      */
/* the DESCRIBE statement.)                                   */
/* - If the data type is decimal (column either nullable or   */
/*   non-nullable), change the type to float because it is    */
/*   easier to display float. Database Manager will convert   */
/*   the decimal data to float format.                        */
/* - If the type is LONG VARCHAR, just allocate memory for    */
/*   4000 characters.                                         */
/* - If the type is VARCHAR or LONG VARCHAR, 2 extra bytes of */
/*   storage are needed because FETCH passes back the current */
/*   length in the first two bytes pointed to by 'sqldata.'   */

 for (i = 0; i<sqlda->sqld && rc == 0; i++)
   {

   adjust = 0;
   if (sqlda->sqlvar[i].sqltype == SQL_TYP_DECIMAL)
     {
     sqlda->sqlvar[i].sqltype = SQL_TYP_FLOAT;
     sqlda->sqlvar[i].sqllen = 8;
     }
```

# QUERY.SQC (CONT'D)

```
    if (sqlda->sqlvar[i].sqltype == SQL_TYP_NDECIMAL)
      {
      sqlda->sqlvar[i].sqltype = SQL_TYP_NFLOAT;
      sqlda->sqlvar[i].sqllen = 8;
      }
    if ((sqlda->sqlvar[i].sqltype == SQL_TYP_VARCHAR) ||
        (sqlda->sqlvar[i].sqltype == SQL_TYP_NVARCHAR))
      {
      adjust = 2;
      }
    if ((sqlda->sqlvar[i].sqltype == SQL_TYP_LONG) ||
        (sqlda->sqlvar[i].sqltype == SQL_TYP_NLONG))
      {
      sqlda->sqlvar[i].sqllen = LONG_DISP_LEN;
      adjust = 2;
      }

    sqlda->sqlvar[i].sqldata = malloc(sqlda->sqlvar[i].sqllen + adjust);
    sqlda->sqlvar[i].sqlind  = malloc(sizeof(short));

   /* If error occurred while getting data storage             */
   /* (i.e. pointer returned as NULL), set return code         */
   /* to error condition.                                      */
   /* Else, increment the number of select list variables      */
   /* for which memory has been obtained.                      */

    if (sqlda->sqlvar[i].sqldata == NULL ||
        sqlda->sqlvar[i].sqlind  == NULL   )
      {
      rc = 965;
      }
    else
      {
      valid_vars++;
      }
    }

 /*****************************************************************/
 /*  If No Errors - FETCH the Data From the Results Table         */
 /*****************************************************************/

  if (rc == 0)
    {
    rc = fetch();
    }
  }

else

  {

 /*****************************************************************/
 /*  Not a SELECT Statement - EXECUTE the PREPARED Statement      */
 /*****************************************************************/

  if (sqlca.sqlcode >= 0)
    {
    EXEC SQL EXECUTE s1;
    }

  rc = (int)sqlca.sqlcode;

  }
```

# QUERY.SQC (CONT'D)

```
/*********************************************************************/
/*  Statement Execution Complete - Check for SQL Error               */
/*********************************************************************/

  if (rc != 0)
    {
      printf("\nError:  sqlcode = %ld.\n", sqlca.sqlcode);
      rc = sqlaintp(msgbuf,BUFSIZE,0,&sqlca);

      if (rc < 0)               /* message retrieve err*/
      {
          printf ("SQLAINTP ERROR. Return code = %d\n",rc);
      }

      if (rc > 0)                /* error message return*/
      {
          printf ("%s\n",msgbuf);
      }
    }

  /*********************************************************************
  /*  Prompt for Another SQL Statement                                 */
  /*********************************************************************/

       printf("SQL stmt: ");
       gets(prep_string);

       } /* End While Loop */

  /*********************************************************************
  /*  Stop Using Database and Return                                   */
  /*********************************************************************

     sqlestpd(&sqlca);

     } /* End Successful Start */

  return(rc);

} /* End MAIN */

/*****************************************************************************
/*  FETCH PROCEDURE FOR SELECT STATEMENTS                                    */
/*                                                                           */
/*   This procedure takes a PREPAREd SELECT statement (accessed with         *
/*   a global PREPAREd statement name), and performs the steps               */
/*   to fetch and print out all rows resulting from the SELECT.              */
/*   This procedure goes through two loops:                                  */
/*    -- the outer loop reads a row of data, until all rows                  */
/*       resulting from the SELECT are read.                                 *
/*    -- the inner loop prints out each value in a row                       */
/*       according to its data type.                                         */
/*                                                                           */
/*****************************************************************************/

static int fetch()

{
  int  rc = 0;
  long i, rec;
  unsigned short vlen;
```

# QUERY.SQC (CONT'D)

```
/*****************************************************************/
/*  OPEN the Cursor                                              */
/*****************************************************************/

 EXEC SQL OPEN C1;
 rc = (int)sqlca.sqlcode;
 rec = 0;

/* If no error from opening the cursor, read a row of data,        */
/* process the SQLDA, and display the data, until all rows are read.*/
/* (The end of the data will be detected by an sqlca.sqlcode =     */
/* SQL_RC_W100, which is 'end of file.')                           */

 if (rc == 0)
   {

   for ( sqlca.sqlcode = 0; sqlca.sqlcode == 0; )
     {

/*****************************************************************/
/*  FETCH a Row of Data Using the Full SQLDA                     */
/*****************************************************************/

   EXEC SQL FETCH c1 USING DESCRIPTOR :*sqlda;

/* If no error from fetch, increment the row counter              */
/* and print out the row number and each value in the row.        */
/* For each value, nullable and non-nullable entries will         */
/* be handled separately.                                         */

   if (sqlca.sqlcode == 0)
     {

     rec++;
     printf("\n%4ld.", rec);

     for (i = 0; i<sqlda->sqld; i++)
       {

       printf(" [");

       switch(sqlda->sqlvar[i].sqltype)
         {

         case SQL_TYP_FLOAT:
              printf("%8f", *((double *)sqlda->sqlvar[i].sqldata));
              break;

         case SQL_TYP_NFLOAT:
              if (*sqlda->sqlvar[i].sqlind < 0)
                {
                printf("<null>");
                }
              else
                {
                printf("%8f", *((double *)sqlda->sqlvar[i].sqldata));
                }
              break;
```

# QUERY.SQC (CONT'D)

```
        case SQL_TYP_INTEGER:
             printf("%14ld", *((long *)sqlda->sqlvar[i].sqldata));
             break;

        case SQL_TYP_NINTEGER:
             if (*sqlda->sqlvar[i].sqlind < 0)
               {
               printf("<null>");
               }
             else
               {
               printf("%14ld", *((long *)sqlda->sqlvar[i].sqldata));
               }
             break;

        case SQL_TYP_SMALL:
             printf("%8d", *((short *)sqlda->sqlvar[i].sqldata));
             break;

        case SQL_TYP_NSMALL:
             if (*sqlda->sqlvar[i].sqlind < 0)
               {
               printf("<null>");
               }
             else
               {
               printf("%8d", *((short *)sqlda->sqlvar[i].sqldata));
               }
             break;

/* Notes:                                                          */
/*   -  the length of VARCHAR and LONG strings are in the 1st      */
/*      two bytes of the location pointed to by 'sqldata'.         */
/*      the actual data begins at offset '2'.                      */

        case SQL_TYP_VARCHAR:
        case SQL_TYP_LONG:
             vlen = *((short *)sqlda->sqlvar[i].sqldata);
             printf("%-.*s", vlen, (sqlda->sqlvar[i].sqldata + 2));
             break;

        case SQL_TYP_NVARCHAR:
        case SQL_TYP_NLONG:

             if (*sqlda->sqlvar[i].sqlind < 0)
               {
               printf("<null>");
               }
             else
               {
               vlen = *((short *)sqlda->sqlvar[i].sqldata);
               printf("%-.*s", vlen, (sqlda->sqlvar[i].sqldata + 2));
               }
             break;

/* Remaining types not covered above.  These are all the fixed    */
/* length character string types.  Note that Date, Time, and      */
/* Timestamp are handled as fixed length character strings upon   */
```

# QUERY.SQC (CONT'D)

```
/* output from the database.                                          */

         default:
              if ((sqlda->sqlvar[i].sqltype % 2) == 0)
                {
                printf("%-.*s", sqlda->sqlvar[i].sqllen,
                                 sqlda->sqlvar[i].sqldata);
                }
              else
                {
                if (*sqlda->sqlvar[i].sqlind < 0)
                  {
                  printf("<null>");
                  }
                else
                  {
                  printf("%-.*s", sqlda->sqlvar[i].sqllen,
                                   sqlda->sqlvar[i].sqldata);

                  }
                }
              break;

         } /* end switch */

       printf("]\n     ");

       } /* end for */
     } /* end if  */
   } /* end for  */

  rc = (int)sqlca.sqlcode;

 /* If cursor should be closed:                                       */
 /*   If there were no rows fetched, print out a message.             */
 /*   Else print out 'end of data.'                                   */
 /* Close the cursor.                                                 */
  if ( rc == SQL_RC_W100 )
    {
    if (rec == 0)
      {
      printf("Sorry...there are no records that satisfy this query\n");
      }
    else
      {
      printf("\n     End of Data\n\n");
      }

    EXEC SQL CLOSE C1;
    rc = (int)sqlca.sqlcode;

    }

  }

return(rc);

} /* End FETCH */
```

# RENAMEDB.SQC

## Change Local Name for a Database

```
/************************************************************************/
/*  PROGRAM NAME: renamedb.sqc                                          */
/*                                                                      */
/*  PURPOSE: Change the local name for a database                       */
/*                                                                      */
/*  FORMAT:  renamedb oldalias newalias                                 */
/*                                                                      */
/*  INPUT:   oldalias       -> current local name for database          */
/*           newalias       -> new local name for database              */
/*                                                                      */
/*  OUTPUT:  Old alias removed, new alias created in system directory   */
/*                                                                      */
/*  PRECOMPILE: any database, delay bind                                */
/*                                                                      */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                                   */
/*                                                                      */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa               */
/*                                                                      */
/*  BIND:    Not necessary                                              */
/*                                                                      */
/*  NOTES:   Database Manager does not have to be started for this      */
/*           program to run.                                            */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                 constant definitions                                 */
/************************************************************************/

#include <stdio.h>                      /* Standard C I/O functions    */
#include <string.h>                     /* C string manipulation fns   */

#include <sqlenv.h>                     /* SQL Environment include     */
#include <sql.h>                        /* SQL constants & err msg fn  */

/************************************************************************/
/*                 Global data elements                                 */
/************************************************************************/

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area     */
char my_dbname[9] = "";
char my_nodename[9] = "";
char my_comment[31] = "";/

/************************************************************************/
/*                 forward declare of routines                          */
/************************************************************************/

void errmsg(void);                      /* error message routine       */
void helpmsg(void);                     /* print out user help         */

/************************************************************************/
/*                   M A I N   R O U T I N E                            */
/************************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{
```

# RENAMEDB.SQC (CONT'D)

```
struct sqledinfo *myinfo;
int rc;
int match = 1;
unsigned short handle, entries, i;
unsigned char drive = '0';                                    /*system dir*/

  if ( (argc >= 2 && *argv[1] == '?') || (argc < 3) )
     {
     helpmsg();                          /*   print out help          */
     return(0);                          /*   leave program           */
     }

  rc = sqledops(drive,&handle,&entries,&sqlca);        /* open directory  */

  if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)           /* error???        */
      errmsg();                                        /* yes - report it */
  else                                                 /* no - scan dir   */

    {
    for (i=1; (i<=entries && match !=0); i++)

        {
        sqledgne(handle,&myinfo,&sqlca);       /* get directory entry     */

        match = strnicmp(argv[1],myinfo->alias,strlen(argv[1]));
        }

    if (match == 0)                                    /* alias found ?   */

       {                                               /* yes, process    */
       rc = sqleuncd(argv[1],&sqlca);                  /* uncatalog alias */

       if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)      /* error?          */
          errmsg();                                    /* yes, report it  */
       else
          {
          strncpy(my_dbname,myinfo->dbname,strcspn(myinfo->dbname," "));

          if ( 0 != strncmp("        ",myinfo->nodename,SQL_NNAME_SZ))
             strncpy(my_nodename,myinfo->nodename,SQL_NNAME_SZ);

          strncpy(my_comment,myinfo->comment,SQL_CMT_SZ);/

          rc = sqlecatd(my_dbname,
                        argv[2],
                        myinfo->type,
                        my_nodename,
                        myinfo->drive[0],
                        my_comment,
                        myinfo->com_codepage,
                        &sqlca);

          if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)   /* error?          */
             errmsg();                                 /* yes, report it  */
          else/
             printf("\nDone!\n");                     /* successful      */

          } /* end - uncatalog OK */

       } /* end - alias found */
```

# RENAMEDB.SQC (CONT'D)

```
    else                                                    /* alias not found */

       printf("\nLocal name \"%s\" not found on your system.\n",argv[1]);
    sqledcls(handle,&sqlca);                    /* close directory scan    */

    }  /* end - entries found */

  return(rc);

}  /* end main */

/*****************************************************************************/
/*                            E R R M S G                                    */
/*****************************************************************************/

void errmsg()
{
#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];

  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);          /* get error msg text      */

  printf("\n%s\n",buffer);                        /* print msg text          */

  return;

}   /* end errmsg */

/*****************************************************************************/
/*                           H E L P M S G                                   */
/*****************************************************************************/
void helpmsg()
{

  printf("RENAMEDB changes the local name for a database\n\n");
  printf("FORMAT: RENAMEDB oldalias newalias\n\n");
  printf("WHERE:  oldalias      -> existing local name of database\n");
  printf("        newalias      -> new local name for database\n");

  return;

}  /* end helpmsg */
```

# UNCATDB.SQC

## Remove Entries from the Database Directory

```
/*****************************************************************/
/*  PROGRAM NAME: uncatdb.sqc                                    */
/*                                                               */
/*  PURPOSE: Remove entries from the system databse directory    */
/*                                                               */
/*  FORMAT:  uncatdb dbname                                      */
/*                                                               */
/*  INPUT:   dbname          -> reference name for database      */
/*                                                               */
/*  OUTPUT:  Entry for requested database removed from system    */
/*           directory                                           */
/*                                                               */
/*  PRECOMPILE: any database, delay bind                         */
/*                                                               */
/*  COMPILE: /AL /FPa /W2 /DLINT_ARGS                            */
/*                                                               */
/*  LINK:    /NOD sql_dyn+sql_stat+doscalls+llibcp+llibfa        */
/*                                                               */
/*  BIND:    Not necessary                                       */
/*                                                               */
/*  NOTES:   Database Manager does not have to be started for this */
/*           program to run.                                     */
/*                                                               */
/*****************************************************************/

/*****************************************************************/
/*                  constant definitions                         */
/*****************************************************************/

#include <stdio.h>                      /* Standard C I/O functions   */
#include <string.h>                     /* C string manipulation fns  */

#include <sqlenv.h>                     /* SQL Environment include    */
#include <sql.h>                        /* SQL constants & err msg fn */

/*****************************************************************/
/*                  Global data elements                         */
/*****************************************************************/

EXEC SQL INCLUDE SQLCA;                 /* SQL Communications area    */

/*****************************************************************/
/*                  forward declare of routines                  */
/*****************************************************************/

void errmsg(void);                      /* error message routine      */
void helpmsg(void);                     /* print out user help        */

/*****************************************************************/
/*                    M A I N   R O U T I N E                    */
/*****************************************************************/
int main(argc, argv)
int argc;
char *argv[];
{
```

# UNCATDB.SQC (CONT'D)

```
int rc;

  if ( (argc >= 2 && *argv[1] == '?') || (argc < 2) )
     {
     helpmsg();                          /*   print out help          */
     return(0);                          /*   leave program           */
     }

  rc = sqleuncd(argv[1],&sqlca);

  if (rc == 0 && sqlca.sqlcode != SQL_RC_OK)          /* error???        */

      errmsg();                                       /* yes - report it */

  else

      printf("\nDone!\n");

  return(rc);

}  /* end main */
/************************************************************************
/*                          E R R M S G                                 */
/************************************************************************/
void errmsg()
{

#define BUFSIZE 512
#define WIDTH 75
unsigned char buffer[BUFSIZE];

  sqlaintp(buffer,BUFSIZE,WIDTH,&sqlca);          /* get error msg text      */

  printf("\n%s\n",buffer);                        /* print msg text          */

  return;/

}   /* end errmsg */

/************************************************************************/
/*                          H E L P M S G                               */
/************************************************************************/

void helpmsg()
{
printf("UNCATDB removes the system directory entry for requested database\n\n");
printf("FORMAT: UNCATDB dbname\n\n");
printf("WHERE:  dbname   -> reference name of database whose directory entry\n");
printf("                    is to be removed\n");/

return;/

}  /* end helpmsg */
```

# Index


Access plan, 19, 122
  definition of, 28
Action bar, 154
Alias, 140, 141
Applheapsz, 56
Application
  batch vs. interactive, 114, 115
  components of database, 10
  database installation, 85, 119
  database vs. non-database, 8
  design considerations, 111–119, 180–185
  installation, 84–87
  non-database, 8
  user interface, 116–117
Application Support routines, 13
  sqlabind, 86, 127–128, 134
  sqlaintp, 76
  sqlaintr, 78, 115
Archival logging, 31
ASC, 200
AUTHID,
  *See* Authorization ID
Authorization ID, 29, 60, 112, 175
Automatic storage class, 69
AVG function, 93

Backup database, 190
Base table, 20
BEGIN DECLARE SECTION statement, 68–71
Bind file, 125, 126–128
Binder, 4
Binding, 86, 125–128, 190
  bind file, 125, 126–128
  effect on design, 117–119
  after generating statistics, 134
  from a program, 89
  timestamp error, 89
Body area, 155
Buffer pool, 45, 52
Buffpage, 53

Callerac, 132–133
CATALGDB.SQC, 145–148
Catalog Tables, 19, 88, 134
  definition of, 30–31
  list of, 30
  for storing access plans, 28
Cataloging, 62
Change database comment, 193
CHAR data type, 20–23, 160
  with dynamic SQL, 104, 106
  limits, 23
  vs. VARCHAR, 22–23
Codepage, 147
Collection name, 29, 60, 112
Commit, 34–35, 44–45, 52, 54, 55, 188, 202
COMMIT statement, 34–35, 44–45
Compiling
  with cc command, 128
  with cl command, 129
  type checking, 128
Concurrency
  designing for, 41–42, 114–115, 184
Configuration parameters
  applheapsz, 56
  buffpage, 53
  codepage, 147
  country, 147
  database, 51
  Database Services, 48
  dbheap, 56
  dlchktime, 52
  locklist, 39, 52
  logext, 55
  logfile, 54
  logmaxext, 55
  maxappls, 56
  maxfilop, 53
  maxtotfilop, 53
  numdb, 49
  purpose, 31–32, 48–57

Configuration parameters—(*continued*)
release, 147
softmax, 55
sortheap, 56
sqlnseg, 49
Configuration routines, 12, 131–132, 147–148
sqlfeudb, 147–148
sqlfrdbc, 147–148
sqlfrsys, 147–148
sqlfusys, 147–148
sqlfxdbc, 147–148
sqlfxsys, 147–148
Consistency
definition of, 33
maintained by recovery, 46
role of locking, 35
role of logging, 35–36
role of transactions, 34–35
Conversion deadlock, 42
COUNT function, 93
Country code, 147
Create
database, 59–60, 139–140, 192
index, 85
table, 85, 198–202
view, 197–198
CREATE INDEX statement, 85
CREATE TABLE statement, 85
CREATEDB.SQC, 139–140
Ctrl+Break, 77–78, 115
Cursors
with dynamic SQL, 98–99, 109–110
open, 88
repositioning, 113
with static SQL, 87–88
using, 113
WITH HOLD, 96, 114

Data accuracy
*See* Data conversions
Data conversions, 24, 69, 91
Data Definition statements, 14
Data exchange, 134–138, 203
Data Manipulation statements, 14
Data required
*See* NOT NULL
Data sharing, 8
Data types, 160
C data types, 72
choosing VARCHAR vs. CHAR, 22–23
for columns, 20, 160
with dynamic SQL, 104
limits, 23
overhead, 23
user defined, 71
Database
automatic selection with Query Manager, 175
backup, 190
buffer pool, 45, 53
codepage, 147
comment, 193
configuration file, 19, 31
configuration parameters, 51
country code, 147
create, 59–60, 192
file structure, 58
installation by application, 85, 119
name, 58, 192
number per OS/2 process, 59, 111–112
number per OS/2 volume, 59, 111–112
password, 192
release, 147
restore, 191
sharing, 67
size of, 58–59
for testing applications, 87
Database Manager
components of, 3–7
directory structure, 60
methods of use, 3
starting, 65–66
use of OS/2, 58–62
Database name, 58, 192
Database Services, 3–4
Application Support routines, 13
configuration parameters, 48
Configuration routines, 12
data objects, 19–29
Environment routines, 11
function calls, 4, 11
supporting objects, 19, 30–32
Utility routines, 12
DATE data type, 20, 160, 172
with dynamic SQL, 104
input by program, 91
limits, 23
retrieved by program, 91
Dbheap, 56
DBIMPORT.SQC, 133, 134, 135
DBLIST.SQC, 142–144
DBSTRUCT.SQC, 88–89, 112
DB2, 203
Deadlock, 42, 52, 115–116
Deadlock cycle, 42

Deadlock detector, 42–43, 52
Debugging, 129–130
DECIMAL data type, 20, 160
  with dynamic SQL, 104, 106
  limits, 23
  with static SQL, 87–88
  vs. FLOAT, 24
DECLARE CURSOR statement, 87–88, 96, 98, 109–110
Declaring cursors
  with dynamic SQL, 98–99, 109–110
  with static SQL, 87–88
  WITH HOLD, 96, 114
DEL, 195, 199
Delete
  row, 93–94
DELETE statement, 93–94
DESCRIBE statement, 102, 105–106
Design considerations, 111–119, 180–185
Directory structure, 60–61
Dlchktime, 52
Drop database, 140–141
Duplicate rows
  avoiding, with indexes, 25
Dynamic SQL, 95–110
  fixed list SELECTs, 98–99
  non-SELECT statements, 97
  parameter markers, 97–98
  performance considerations, 96
  steps to execute, 96–97
  varying list SELECTS, 99–105

END DECLARE SECTION statement, 68–71
Environment routines, 11, 138–146
  sqlecatd, 145
  sqlecred, 139–140
  sqledcls, 142
  sqledgne, 142
  sqledops, 142
  sqledrpd, 140–141
  sqleisig, 77–78
  sqlerest, 66–67
  sqlestar, 65–66
  sqlestop, 66, 67
  sqlestrd, 66–67
  sqleuncd, 146
Erase
  database
    *See* Drop database
  row
    *See* Delete, row
ERASEDB.SQC, 140–141
Error
  messages, 76–77
  processing, 71–77
  returned in SQLCA, 72–73
Escalation, 39
EXECUTE IMMEDIATE statement, 97
EXECUTE statement, 97–98
Export
  exchange formats, 135, 137–138
  queries, 177, 169
  sqluexp, 135
  table, 194–198
Extrn storage class, 69

FETCH statement, 88, 89–92, 96–97
File
  name, 58, 60
  number of, 53
FILLTB.SQC, 86–87, 119
FLOAT data type, 20, 91, 160, 172
  with dynamic SQL, 104, 106
  limits, 23
  vs. DECIMAL, 24
Floating point, 24, 172
FOR BIT DATA, 24, 160
Forward recovery, 31
Function calls
  *See* Database Services

GRANT statement, 32
Granted group, 40

Hierarchy of locks, 38–39
High-level languages
  reasons for using, 5
  use in database application, 10
Host variables
  C data types, 70
  containing SQL statements, 97
  declaring, 68–71
  in dynamic SQL, 97
  for nullable columns, 69
  pointers as, 71, 90
  restrictions, 70–71
  in static SQL, 96
  storage classes, 69
  structures as, 71, 90

ID for lists, 29, 60, 112, 175–176
Import, 54
  exchange formats, 135, 137–138, 199
  method of database installation, 86, 119

Import—(*continued*)
queries, 177, 169
sqluimp, 135
table, 199–202
Include files
sqlca.h, 72, 79
sqlcodes.h, 79
sqlda.h, 79
sqlenv.h, 79
sqlutil.h, 79
sql.h, 79
INCLUDE SQLCA statement, 72
INCLUDE SQLDA statement, 100–102
Index, 84
for concurrent access, 26
cost of, 26–27
creating, 25
definition of, 24
effect on bulk updates, 27
enforcing uniqueness with, 25
file name for, 58
for primary keys, 25
use of, 24
Index scan, 26, 37–38
INSERT statement, 85–86
Inserting data rows, 85–86
Installation
file structure, 60
INSTALL.SQC, 86–87, 119
INTEGER data type, 20, 160
with dynamic SQL, 104
limits, 23
IS lock, 36–37
IX lock, 36–37
IXF, 194, 199

Keys
*See* Index

Linking, 129
Locking, 35–44, 52
Locklist, 39, 52
Locks
acquired by single transaction, 37–38
compatibility between transactions, 40
configuration options, 52
conversion deadlock, 42
conversion of, 38
deadlock, 42, 115–116
determining type of, 36–40
duration, 114
effects on concurrency, 41
escalation, 39
hierarchy of, 38–39
IS lock, 36–37
IX lock, 36–37
relationship to indexes, 26
removing, 38
S lock, 36
SIX lock, 38
waiting for, 35, 115
when to acquire, 114–115
X lock, 36
Log, 35–36, 54, 58
definition of, 31, 54
extents, 45, 54
log full, 116, 202
managing, 45–46
use during recovery, 46, 54
Log full, 202
Log record, 44
Logext, 54
Logfile, 54
LOGFILE.SQC, 147
Logging, 44–47
Logmaxext, 55
LONG VARCHAR data type, 20, 160
with dynamic SQL, 104–106
limits, 23

MAX function, 93
Maxappls, 56
Maxfilop, 53
Maxtotfilop, 53
Media failures, 31
Menus, 152
main, 155
pop-ups, 155
pull-downs, 155
MIN function, 93

Naming conflict, 60, 112
Naming objects, 29–30, 112
NOT NULL, 23, 93, 160
NULL, 93
Null indicator, 69, 88, 93
Nullable, 93
Numdb, 49

Objects
access plans, 28
Catalog Tables, 30–31
configuration parameters, 31
Database Services, 19–30

defining in a program, 84–87
indexes, 28–29
naming, 29–30
naming conflict, 30, 60
Query Manager, 151
recovery log, 31
tables, 20–24
views, 27–28
OPEN cursor statement, 88
Optimizer, 41–42, 85
ORDER BY clause, 20, 113, 170
Original name, 62, 192
OS/2
Extended Edition, 3
files, 53–54
process, 111–112
shared segments, 50
Standard Edition, 3
threads, 111–112

Packed decimal, 91, 106–107
Panels, 179–188
Parameter markers
with dynamic SELECTs, 99
in dynamic SQL, 97–98
Passwords, 32, 192, 191
Performance, 52, 53, 184
concurrency vs. performance, 114–115
cost of indexes, 26
improving, 132–134
Precompilation, 13–14, 124–128
with bind, 124–126
without bind, 126–128
delimiters, 67–68
Precompiler, 4
directives, 126
SQLPREP, 124–125
PREPARE statement, 97–98
Primary keys, 25, 42
Procedures, 173–175, 179
Process, 111–112
Profiles, 177
Prompted query, 165
Convert to SQL, 166–167
limitations, 165
Show SQL, 166

QMF, 203
Qualifier, 29, 60, 112, 175
Query, 162–178
Query Manager, 3, 14
objects, 151
profile, 177
tasks performed with, 6–7
variables, 166, 186
Query Manager menus
Databases Action pull-down, 158
Databases Main Menu, 155
Databases password prompt, 159
generated SQL, 169
Main Selection menu, 160
menu roadmap, 157
OS/2 Desktop, 153
Prompted Query menu, 167
report usage code specification, 171
report with multiple usage codes, 171
sample report from STAFF, 168
Tables and Views, 193
QUERY.SQC, 108–110

Reading data
*See* Retrieving data
Record number, 113
Recovery, 35–36, 45, 46–47
soft checkpoint, 47, 55
Recovery log, 19, 31
*See also* Log
Reference name, 62, 192
Release, 147
Reorganize table, 67
Repeatable read, 34–35
Reports
changing formats, 170–172
control breaks, 167
heading text, 172
multiple summary functions, 170
ordering the output, 170
Restore database, 191
Result table, 20
Retrieving data, 87–92
character strings, 90
DATE, 91–92
DECIMAL, 91, 106–107
FLOAT, 91
multiple rows, 87–88, 113
ORDER BY clause, 113
single row, 87
TIME, 91–92
TIMESTAMP, 91–92
REVOKE statement, 32
Rollback, 54
ROLLBACK statement, 34–35, 44–45
Row Buffer, 176

S lock, 36
SAMPLE database, 60, 142
Sample programs
  CATALGDB.SQC, 145–148
  CREATEDB.SQC, 139–140
  DBIMPORT.SQC, 133, 134, 135
  DBLIST.SQC, 142–144
  DBSTRUCT.SQC, 88–89, 112
  ERASEDB.SQC, 140–141
  FILLTB.SQC, 86–87, 119
  INSTALL.SQC, 86–87, 119
  LOGFILE.SQC, 147
  QUERY.SQC, 108–110
  UNCATDB.SQC, 146
Scientific data
  *See* Floating point
Security, 32
SELECT statement, 87
  fixed list, 98–99
  INTO clause, 87
  multiple rows, 87–88
  result table, 20
  SELECT * in programs, 90–91
  single row, 87
  varying list, 99–105
Shared segments, 49–51, 56
SIX lock, 38
SMALLINT data type, 20, 160
  with dynamic SQL, 104
  limits, 23
Soft checkpoint, 47, 55
Softmax, 55
Sortheap, 56
SQL, 3–4, 10, 13
  column data types, 20
  column functions, 91
  dynamic, 95–110
  set orientation, 113
  statement types, 14
  static vs. dynamic, 14, 95
SQL Communications Area
  *See* SQLCA
SQL Descriptor Area
  *See* SQLDA
SQL query, 164–165, 168, 198, 201
SQL statements
  BEGIN DECLARE SECTION, 68–71
  COMMIT, 34–35, 44–45
  CREATE INDEX, 85
  CREATE TABLE, 85
  DECLARE CURSOR, 87–88, 96, 98, 109–110
  DELETE, 93–94
  DESCRIBE, 102, 105–106
  END DECLARE SECTION, 68–71
  EXECUTE, 97–98
  EXECUTE IMMEDIATE, 97
  FETCH, 88, 89–92, 96–97
  GRANT, 32
  INCLUDE SQLCA, 72
  INCLUDE SQLDA, 100–102
  INSERT, 85–86
  non-dynamic statements, 96
  OPEN cursor, 88
  PREPARE, 97–98
  in program text, 67–68
  REVOKE, 32
  ROLLBACK, 34–35, 44–45
  SELECT, 87
  UPDATE, 92
  WHENEVER, 73–76
Sqlabind, 86, 127–128, 134
Sqlaintp, 76
Sqlaintr, 78, 115
SQLBIND, 127
SQLCA, 72
  declaring, 72
  with function calls, 73
  with SQL statements, 73
  structure, 72
Sqlca.h, 72, 79
Sqlcodes.h, 79
SQLDA
  allocating space for, 101
  description, 102–103
  with dynamic SELECTs, 99–100
  SQLDASIZE macro, 101
  structure, 100
SQLDASIZE macro, 101
Sqlda.h, 79
Sqlecatd, 145
Sqlecred, 139–140
Sqledcls, 142
Sqledgne, 142
Sqledops, 142
Sqledrpd, 140–141
Sqleisig, 77–78
Sqlenv.h, 79
Sqlerest, 66–67
Sqlestar, 65–66
Sqlestop, 66, 71
Sqlestrd, 66–67
Sqleuncd, 146
Sqlfeudb, 147–148
Sqlfrdbc, 147–148
Sqlfrsys, 147–148
Sqlfusys, 147–148

Sqlfxdbc, 147–148
Sqlfxsys, 147–148
SQLLIB, 198, 202
Sqlnseg, 49
SQLPREP, 124–125
SQLTYPE
  listing of values, 104
Sqluexp, 135
Sqluimp, 135
Sqlureor, 133
SQLUSER, 29, 60, 112, 176
Sqlustat, 134
Sqlutil.h, 79
Sql.h, 79
SQL/DS, 203
Start Programs window, 152
STARTDBM
  from a command file, 65–66
Starting the Query Manager, 153
Static staorage class, 69
Statistics, 134
  executing from program, 134
  sqlustat, 134
  when to run, 134
Structured Query Language
  *See* SQL
SUM function, 91
System directory, 60, 141, 142–143
S/370 data exchange, 203

Table, 19
  base table, 20
  copy from QMF, 203
  create view for, 197–198
  creating, 20
  definition of, 20
  export, 194–198
  file name for, 58
  import, 199–202
  limiting size of, 59
  result table, 20
Table scan, 26, 37–38, 41
Task Manager window, 152
Testing, 60, 87
Threads, 111–112
TIME data type, 20, 160, 172
  with dynamic SQL, 104
  input by program, 91
  limits, 23
  retrieved by program, 91
TIMESTAMP data type, 20, 160, 172
  with dynamic SQL, 104
  input by program, 91
  limits, 23
  retrieved by program, 91
Timestamp error, 89
Title line, 154
Transaction
  commit
    *See* Commit
  definition of, 34–35, 113
  length of, 35, 114
  multiple threads in, 112
  rollback
    *See* Rollback
Transaction management, 9

UNCATDB.SQC, 146
UNIQUE, 25
UPDATE statement, 92
Updating rows, 92–93
User interface
  design considerations, 116–117
Utilities
  backup database, 190
  export table, 194–198
  import table, 199–202
  restore database, 191
  S/370 data exchange, 203
Utility routines, 12, 131–138
  callerac, 132–133
  sqluexp, 135
  sqluimp, 135
  sqlureor, 133
  sqlustat, 134

VARCHAR data type, 20, 160
  with dynamic SQL, 104, 106
  limits, 23
  vs. CHAR, 22–23
Variables, 166, 186
View, 19
  create, 197–198
  definition of, 27
  with indexes, 28
  range checking with, 28
  updateable, 28
Volume directory, 60, 141, 142–143

WHENEVER statement, 73–76
WITH CHECK OPTION, 28
Write ahead logging, 44–45
WSF, 195, 199

X lock, 36

3 9001 02811 4778